Javascript: Object Oriented Programming

Build sophisticated web applications by mastering the art of Object-Oriented Javascript

Packt>

LEARNING PATH

Javascript: Object Oriented Programming

Build sophisticated web applications by mastering the art of Object-Oriented Javascript

A course in three modules

Packt

BIRMINGHAM - MUMBAI

Javascript: Object Oriented Programming

Published on: August 2016

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78712-359-5

www.packtpub.com

Credits

Authors
Ved Antani
Gastón C. Hillar
Stoyan Stefanov
Kumar Chetan Sharma

Reviewers
Ivano Malavolta
Róman Joos
Hugo Solis
Kumar Chetan Sharma
Alex R. Young

Content Development Editor
Deepti Thore

Production Coordinator
Arvindkumar Gupta

Preface

It may seem that everything that needs to be written about JavaScript has already been written. However, JavaScript is changing rapidly. ECMAScript 6 has the potential to transform the language and how we code in it. Node.js has changed the way we write servers in JavaScript. Newer ideas such as React and Flux will drive the next iteration of the language. If you are already an experienced JavaScript developer, you will realize that modern JavaScript is vastly different from the language that most people have known. Tools are more powerful and slowly becoming an integral part of the development workflow.

Object-oriented programming, also known as OOP, is a required skill in absolutely any modern software developer job. It makes a lot of sense because object-oriented programming allows you to maximize code reuse and minimize the maintenance costs. However, learning object-oriented programming is challenging because it includes too many abstract concepts that require real-life examples to make it easy to understand.

JavaScript has moved from being mostly used in browsers for client-side technologies to being used even on server side.

This course will help you change some common coding practices and empower you by giving you the tools you need for more efficient development. We'll look at implementing these principles to build sophisticated web applications.

We hope that you enjoy this course as much as we enjoyed developing it.

What this learning path covers

Module 1, Mastering Javascript, provides you with a detailed overview of the language's fundamentals and some of the modern tools and libraries, such as jQuery, Underscore.js, and Jasmine.

Module 2 Learning Object-Oriented Programming, helps you to learn how to capture objects from real-world elements and create object-oriented code that represents them.

Module 3, Object-Oriented JavaScript - Second Edition, This module doesn't assume any prior knowledge of JavaScript and works from the ground up to give you a thorough understanding of the language What you need for this learning path. Exercises at the end of the chapters help you assess your understanding.

What you need for this learning path

- A computer with Windows 7 or higher, Linux, or Mac OS X installed.
- The latest version of the Google Chrome or Mozilla Firefox browser.
- A text editor of your choice. Sublime Text, vi, Atom, or Notepad++ would be ideal. The choice is entirely yours
- A computer with at least an Intel Core i3 CPU or equivalent with 4 GB RAM, running on Windows 7 or a higher version, Mac OS X Mountain Lion or a higher version, or any Linux version that is capable of running Python 3.4, and a browser with JavaScript support.
- You will need Python 3.4.3 installed on your computer. You can work with your favorite editor or use any Python IDE that is compatible with the mentioned Python version.
- In order to work with the C# examples, you will need Visual Studio 2015 or 2013.
- You can use the free Express editions to run all the examples. If you aren't working on Windows, you can use Xamarin Studio 5.5 or higher.
- In order to work with the JavaScript examples, you will need web browsers such as Chrome 40.x or higher, Firefox 37.x or higher, Safari 8.x or higher, Internet Explorer 10 or higher that provides a JavaScript console
- You need a modern browser—Google Chrome or Firefox are recommended—and an optional Node.js setup. The latest version of Firefox comes with web developer tools, but Firebug is highly recommended. To edit JavaScript you can use any text editor of your choice.

Who this learning path is for

JavaScript developers looking to enhance their web developments skills by learning object-oriented programming.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at `http://www.packtpub.com`. If you purchased this course elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at `https://github.com/PacktPublishing/Object-oriented-programming-for-JavaScript-developers`. We also have other code bundles from our rich catalog of books, videos, and courses available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Course Module 1: Mastering JavaScript

Module 1

Mastering JavaScript

Explore and master modern JavaScript techniques in order
to build large-scale web applications

1
JavaScript Primer

It is always difficult to pen the first few words, especially on a subject like JavaScript. This difficulty arises primarily because so many things have been said about this language. JavaScript has been the *Language of the Web*—lingua franca, if you will, since the earliest days of the Netscape Navigator. JavaScript went from a tool of the amateur to the weapon of the connoisseur in a shockingly short period of time.

JavaScript is the most popular language on the web and open source ecosystem. `http://githut.info/` charts the number of active repositories and overall popularity of the language on GitHub for the last few years. JavaScript's popularity and importance can be attributed to its association with the browser. Google's V8 and Mozilla's SpiderMonkey are extremely optimized JavaScript engines that power Google Chrome and Mozilla Firefox browsers, respectively.

Although web browsers are the most widely used platforms for JavaScript, modern databases such as MongoDB and CouchDB use JavaScript as their scripting and query language. JavaScript has become an important platform outside browsers as well. Projects such as **Node.js** and **io.js** provide powerful platforms to develop scalable server environments using JavaScript. Several interesting projects are pushing the language capabilities to its limits, for example, **Emscripten** (`http://kripken.github.io/emscripten-site/`) is a **Low-Level Virtual Machine** (**LLVM**)-based project that compiles C and C++ into highly optimizable JavaScript in an **asm.js** format. This allows you to run C and C++ on the web at near native speed.

JavaScript is built around solid foundations regarding, for example, functions, dynamic objects, loose typing, prototypal inheritance, and a powerful object literal notation.

While JavaScript is built on sound design principles, unfortunately, the language had to evolve along with the browser. Web browsers are notorious in the way they support various features and standards. JavaScript tried to accommodate all the whims of the browsers and ended up making some very bad design decisions. These bad parts (the term made famous by Douglas Crockford) overshadowed the good parts of the language for most people. Programmers wrote bad code, other programmers had nightmares trying to debug that bad code, and the language eventually got a bad reputation. Unfortunately, JavaScript is one of the most misunderstood programming languages (`http://javascript.crockford.com/javascript.html`).

Another criticism leveled at JavaScript is that it lets you get things done without you being an expert in the language. I have seen programmers write exceptionally bad JavaScript code just because they wanted to get the things done quickly and JavaScript allowed them to do just this. I have spent hours debugging very bad quality JavaScript written by someone who clearly was not a programmer. However, the language is a tool and cannot be blamed for sloppy programming. Like all crafts, programming demands extreme dedication and discipline.

A little bit of history

In 1993, the **Mosaic** browser of **National Center for Supercomputing Applications** (**NCSA**) was one of the first popular web browsers. A year later, Netscape Communications created the proprietary web browser, **Netscape Navigator**. Several original Mosaic authors worked on Navigator.

In 1995, Netscape Communications hired Brendan Eich with the promise of letting him implement **Scheme** (a Lisp dialect) in the browser. Before this happened, Netscape got in touch with Sun Microsystems (now Oracle) to include Java in the Navigator browser.

Due to the popularity and easy programming of Java, Netscape decided that a scripting language had to have a syntax similar to that of Java. This ruled out adopting existing languages such as Python, **Tool Command Language** (**TCL**), or Scheme. Eich wrote the initial prototype in just 10 days (`http://www.computer.org/csdl/mags/co/2012/02/mco2012020007.pdf`), in May 1995. JavaScript's first code name was **Mocha**, coined by Marc Andreessen. Netscape later changed it to **LiveScript**, for trademark reasons. In early December 1995, Sun licensed the trademark Java to Netscape. The language was renamed to its final name, JavaScript.

How to use this book

This book is not going to help if you are looking to get things done quickly. This book is going to focus on the correct ways to code in JavaScript. We are going to spend a lot of time understanding how to avoid the bad parts of the language and build reliable and readable code in JavaScript. We will skirt away from sloppy features of the language just to make sure that you are not getting used to them—if you have already learned to code using these habits, this book will try to nudge you away from this. There will be a lot of focus on the correct style and tools to make your code better.

Most of the concepts in this book are going to be examples and patterns from real-world problems. I will insist that you code each of the snippets to make sure that your understanding of the concept is getting programmed into your muscle memory. Trust me on this, there is no better way to learn programming than writing a lot of code.

Typically, you will need to create an HTML page to run an embedded JavaScript code as follows:

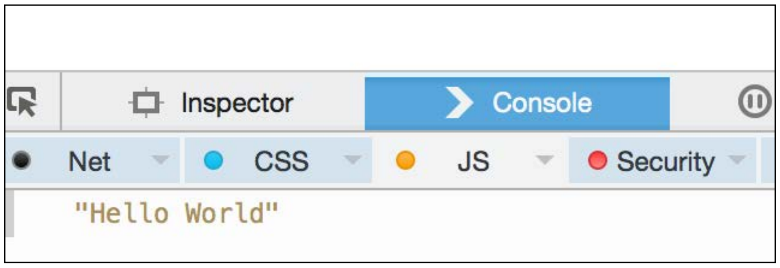
```
<!DOCTYPE html>
<html>
<head>
  <script type="text/javascript" src="script.js"></script>
  <script type="text/javascript">
    var x = "Hello World";
    console.log(x);
  </script>
</head>
<body>
</body>
</html>
```

This sample code shows two ways in which JavaScript is embedded into the HTML page. First, the `<script>` tag in `<head>` imports JavaScript, while the second `<script>` tag is used to embed inline JavaScript.

Downloading the example code

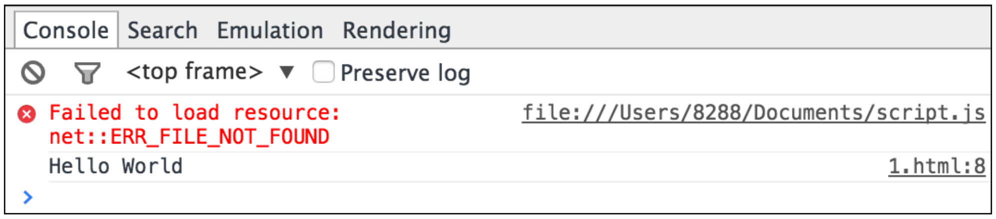
You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can save this HTML page locally and open it in a browser. On Firefox, you can open the **Developer** console (Firefox menu | **Developer** | **Web Console**) and you can see the **"Hello World"** text on the **Console** tab. Based on your OS and browser version, the screen may look different:



```
"Hello World"
```

You can run the page and inspect it using Chrome's **Developer Tool**:



```
Failed to load resource:                file:///Users/8288/Documents/script.js
net::ERR_FILE_NOT_FOUND
Hello World                                                         1.html:8
```

A very interesting thing to notice here is that there is an error displayed on the console regarding the missing `.js` file that we are trying to import using the following line of code:

```
<script type="text/javascript" src="script.js"></script>
```

Using browser developer consoles or an extension such as **Firebug** can be very useful in debugging error conditions in the code. We will discuss in detail the debugging techniques in later chapters.

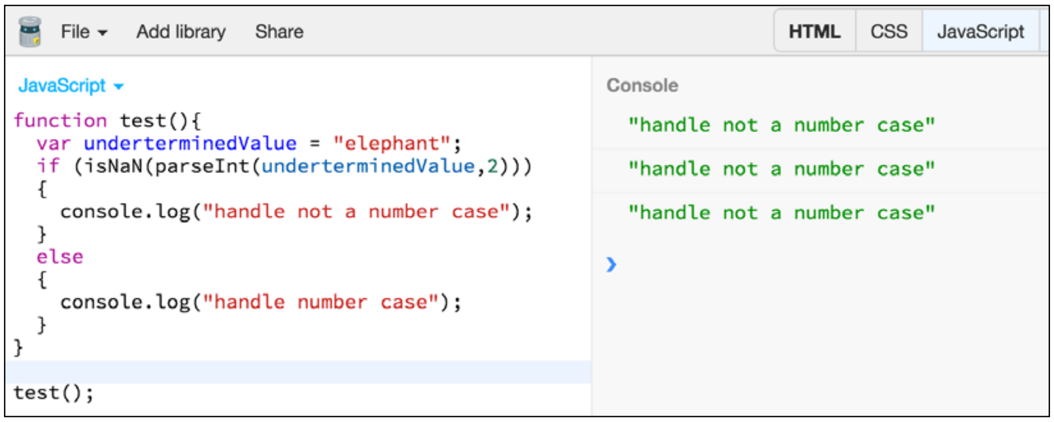
Creating such HTML scaffolds can be tedious for every exercise in this book. Instead, we want to use a **Read-Eval-Print-Loop** (**REPL**) for JavaScript. Unlike Python, JavaScript does not come packaged with an REPL. We can use Node.js as an REPL. If you have Node.js installed on your machine, you can just type `node` on the command line and start experimenting with it. You will observe that Node REPL errors are not very elegantly displayed.

Let's see the following example:

```
EN-VedA:~$ node

>function greeter(){
  x="World"l

SyntaxError: Unexpected identifier

  at Object.exports.createScript (vm.js:44:10)

  at REPLServer.defaultEval (repl.js:117:23)

  at bound (domain.js:254:14)

  …
```

After this error, you will have to restart. Still, it can help you try out small fragments of code a lot faster.

Another tool that I personally use a lot is **JS Bin** (`http://jsbin.com/`). JS Bin provides you with a great set of tools to test JavaScript, such as syntax highlighting and runtime error detection. The following is a screenshot of JS Bin:



```
function test(){
  var underterminedValue = "elephant";
  if (isNaN(parseInt(underterminedValue,2)))
  {
    console.log("handle not a number case");
  }
  else
  {
    console.log("handle number case");
  }
}

test();
```

```
"handle not a number case"
"handle not a number case"
"handle not a number case"
```

Based on your preference, you can pick the tool that makes it easier to try out the code samples. Regardless of which tool you use, make sure that you type out every exercise in this book.

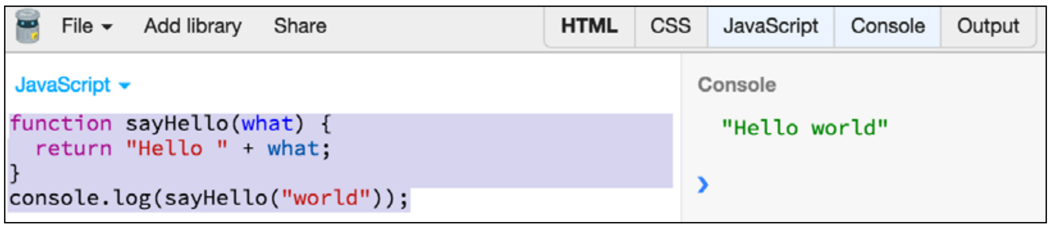
Hello World

No programming language should be published without a customary Hello World program—why should this book be any different?

Type (don't copy and paste) the following code in JS Bin:

```
function sayHello(what) {
  return "Hello " + what;
}
console.log(sayHello("world"));
```

Your screen should look something as follows:



```
function sayHello(what) {
  return "Hello " + what;
}
console.log(sayHello("world"));
```

```
"Hello world"
```

An overview of JavaScript

In a nutshell, JavaScript is a prototype-based scripting language with dynamic typing and first-class function support. JavaScript borrows most of its syntax from Java, but is also influenced by Awk, Perl, and Python. JavaScript is case-sensitive and white space-agnostic.

Comments

JavaScript allows single line or multiple line comments. The syntax is similar to C or Java:

```
// a one line comment

/* this is a longer,
   multi-line comment
 */

/* You can't /* nest comments */ SyntaxError */
```

Variables

Variables are symbolic names for values. The names of variables, or identifiers, must follow certain rules.

A JavaScript variable name must start with a letter, underscore (_), or dollar sign ($); subsequent characters can also be digits (0-9). As JavaScript is case sensitive, letters include the characters *A* through *Z* (uppercase) and the characters *a* through *z* (lowercase).

You can use ISO 8859-1 or Unicode letters in variable names.

New variables in JavaScript should be defined with the **var** keyword. If you declare a variable without assigning a value to it, its type is undefined by default. One terrible thing is that if you don't declare your variable with the var keyword, they become implicit globals. Let me reiterate that implicit globals are a terrible thing—we will discuss this in detail later in the book when we discuss variable scopes and closures, but it's important to remember that you should always declare a variable with the var keyword unless you know what you are doing:

```
var a;      //declares a variable but its undefined
var b = 0;
console.log(b);    //0
console.log(a);    //undefined
console.log(a+b);  //NaN
```

The `NaN` value is a special value that indicates that the entity is *not a number*.

Constants

You can create a read-only named constant with the **const** keyword. The constant name must start with a letter, underscore, or dollar sign and can contain alphabetic, numeric, or underscore characters:

```
const area_code = '515';
```

A constant cannot change the value through assignment or be redeclared, and it has to be initialized to a value.

JavaScript supports the standard variations of types:

- Number
- String
- Boolean
- Symbol (new in ECMAScript 6)
- Object:
    - Function
    - Array
    - Date
    - RegExp
- Null
- Undefined

Number

The **Number** type can represent both 32-bit integer and 64-bit floating point values. For example, the following line of code declares a variable to hold an integer value, which is defined by the literal 555:

```
var aNumber = 555;
```

To define a floating point value, you need to include a decimal point and one digit after the decimal point:

```
var aFloat = 555.0;
```

Essentially, there's no such thing as an integer in JavaScript. JavaScript uses a 64-bit floating point representation, which is the same as Java's double.

Hence, you would see something as follows:

```
EN-VedA:~$ node
> 0.1+0.2
0.30000000000000004
> (0.1+0.2)===0.3
false
```

I recommend that you read the exhaustive answer on Stack Overflow (`http://stackoverflow.com/questions/588004/is-floating-point-math-broken`) and (`http://floating-point-gui.de/`), which explains why this is the case. However, it is important to understand that floating point arithmetic should be handled with due care. In most cases, you will not have to rely on extreme precision of decimal points but if you have to, you can try using libraries such as **big.js** (`https://github.com/MikeMcl/big.js`) that try to solve this problem.

If you intend to code extremely precise financial systems, you should represent $ values as cents to avoid rounding errors. One of the systems that I worked on used to round off the **Value Added Tax** (**VAT**) amount to two decimal points. With thousands of orders a day, this rounding off amount per order became a massive accounting headache. We needed to overhaul the entire Java web service stack and JavaScript frontend for this.

A few special values are also defined as part of the Number type. The first two are `Number.MAX_VALUE` and `Number.MIN_VALUE`, which define the outer bounds of the Number value set. All ECMAScript numbers must fall between these two values, without exception. A calculation can, however, result in a number that does not fall in between these two numbers. When a calculation results in a number greater than `Number.MAX_VALUE`, it is assigned a value of `Number.POSITIVE_INFINITY`, meaning that it has no numeric value anymore. Likewise, a calculation that results in a number less than `Number.MIN_VALUE` is assigned a value of `Number.NEGATIVE_INFINITY`, which also has no numeric value. If a calculation returns an infinite value, the result cannot be used in any further calculations. You can use the `isInfinite()` method to verify if the calculation result is an infinity.

Another peculiarity of JavaScript is a special value called NaN (short for *Not a Number*). In general, this occurs when conversion from another type (String, Boolean, and so on) fails. Observe the following peculiarity of NaN:

```
EN-VedA:~ $ node
> isNaN(NaN);
true
> NaN==NaN;
false
> isNaN("elephant");
true
> NaN+5;
NaN
```

The second line is strange—NaN is not equal to NaN. If NaN is part of any mathematical operation, the result also becomes NaN. As a general rule, stay away from using NaN in any expression. For any advanced mathematical operations, you can use the `Math` global object and its methods:

```
> Math.E
2.718281828459045
> Math.SQRT2
1.4142135623730951
> Math.abs(-900)
900
> Math.pow(2,3)
8
```

You can use the `parseInt()` and `parseFloat()` methods to convert a string expression to an integer or float:

```
> parseInt("230",10);
230
> parseInt("010",10);
10
> parseInt("010",8); //octal base
8
> parseInt("010",2); //binary
2
> + "4"
4
```

With `parseInt()`, you should provide an explicit base to prevent nasty surprises on older browsers. The last trick is just using a `+` sign to auto-convert the `"42"` string to a number, `42`. It is also prudent to handle the `parseInt()` result with `isNaN()`. Let's see the following example:

```
var underterminedValue = "elephant";
if (isNaN(parseInt(underterminedValue,2)))
{
   console.log("handle not a number case");
}
else
{
   console.log("handle number case");
}
```

In this example, you are not sure of the type of the value that the `underterminedValue` variable can hold if the value is being set from an external interface. If `isNaN()` is not handled, `parseInt()` will cause an exception and the program can crash.

String

In JavaScript, strings are a sequence of Unicode characters (each character takes 16 bits). Each character in the string can be accessed by its index. The first character index is zero. Strings are enclosed inside `"` or `'`—both are valid ways to represent strings. Let's see the following:

```
> console.log("Hippopotamus chewing gum");
Hippopotamus chewing gum
> console.log('Single quoted hippopotamus');
Single quoted hippopotamus
> console.log("Broken \n lines");
Broken
 lines
```

The last line shows you how certain character literals when escaped with a backslash \ can be used as special characters. The following is a list of such special characters:

- `\n`: Newline
- `\t`: Tab
- `\b`: Backspace
- `\r`: Carriage return
- `\\`: Backslash
- `\'`: Single quote
- `\"`: Double quote

You get default support for special characters and Unicode literals with JavaScript strings:

```
> '\xA9'
'©'
> '\u00A9'
'©'
```

One important thing about JavaScript Strings, Numbers, and Booleans is that they actually have wrapper objects around their primitive equivalent. The following example shows the usage of the wrapper objects:

```
var s = new String("dummy"); //Creates a String object
console.log(s); //"dummy"
console.log(typeof s); //"object"
var nonObject = "1" + "2"; //Create a String primitive
console.log(typeof nonObject); //"string"
var objString = new String("1" + "2"); //Creates a String object
console.log(typeof objString); //"object"
//Helper functions
console.log("Hello".length); //5
console.log("Hello".charAt(0)); //"H"
console.log("Hello".charAt(1)); //"e"
console.log("Hello".indexOf("e")); //1
console.log("Hello".lastIndexOf("l")); //3
console.log("Hello".startsWith("H")); //true
console.log("Hello".endsWith("o")); //true
console.log("Hello".includes("X")); //false
var splitStringByWords = "Hello World".split(" ");
console.log(splitStringByWords); //["Hello", "World"]
var splitStringByChars = "Hello World".split("");
console.log(splitStringByChars); //["H", "e", "l", "l", "o", " ",
  "W", "o", "r", "l", "d"]
console.log("lowercasestring".toUpperCase()); //"LOWERCASESTRING"
console.log("UPPPERCASESTRING".toLowerCase());
  //"upppercasestring"
console.log("There are no spaces in the end     ".trim());
  //"There are no spaces in the end"
```

JavaScript allows multiline strings also. Strings enclosed within ` (Grave accent—`https://en.wikipedia.org/wiki/Grave_accent`) are considered multiline. Let's see the following example:

```
> console.log(`string text on first line
string text on second line `);
"string text on first line
string text on second line "
```

This kind of string is also known as a template string and can be used for string interpolation. JavaScript allows Python-like string interpolation using this syntax.

Normally, you would do something similar to the following:

```
var a=1, b=2;
console.log("Sum of values is :" + (a+b) + " and multiplication is :"
  + (a*b));
```

However, with string interpolation, things become a bit clearer:

```
console.log(`Sum of values is :${a+b} and multiplication is :
  ${a*b}`);
```

Undefined values

JavaScript indicates an absence of meaningful values by two special values—null, when the non-value is deliberate, and undefined, when the value is not assigned to the variable yet. Let's see the following example:

```
> var xl;
> console.log(typeof xl);
undefined
> console.log(null==undefined);
true
```

Booleans

JavaScript Boolean primitives are represented by `true` and `false` keywords. The following rules govern what becomes false and what turns out to be true:

- False, 0, the empty string (""), NaN, null, and undefined are represented as false
- Everything else is true

JavaScript Booleans are tricky primarily because the behavior is radically different in the way you create them.

There are two ways in which you can create Booleans in JavaScript:

- You can create primitive Booleans by assigning a true or false literal to a variable. Consider the following example:

  ```
  var pBooleanTrue = true;
  var pBooleanFalse = false;
  ```

- Use the `Boolean()` function; this is an ordinary function that returns a primitive Boolean:

    ```
    var fBooleanTrue = Boolean(true);
    var fBooleanFalse = Boolean(false);
    ```

Both these methods return expected *truthy* or *falsy* values. However, if you create a Boolean object using the `new` operator, things can go really wrong.

Essentially, when you use the `new` operator and the `Boolean(value)` constructor, you don't get a primitive `true` or `false` in return, you get an object instead—and unfortunately, JavaScript considers an object as *truthy*:

```
var oBooleanTrue = new Boolean(true);
var oBooleanFalse = new Boolean(false);
console.log(oBooleanTrue); //true
console.log(typeof oBooleanTrue); //object
if(oBooleanFalse){
 console.log("I am seriously truthy, don't believe me");
}
>"I am seriously truthy, don't believe me"

if(oBooleanTrue){
 console.log("I am also truthy, see ?");
}
>"I am also truthy, see ?"

//Use valueOf() to extract real value within the Boolean object
if(oBooleanFalse.valueOf()){
 console.log("With valueOf, I am false");
}else{
 console.log("Without valueOf, I am still truthy");
}
>"Without valueOf, I am still truthy"
```

So, the smart thing to do is to always avoid Boolean constructors to create a new Boolean object. It breaks the fundamental contract of Boolean logic and you should stay away from such difficult-to-debug buggy code.

The instanceof operator

One of the problems with using reference types to store values has been the use of the **typeof** operator, which returns `object` no matter what type of object is being referenced. To provide a solution, you can use the **instanceof** operator. Let's see some examples:

```
var aStringObject = new String("string");
console.log(typeof aStringObject);        //"object"
console.log(aStringObject instanceof String);    //true
var aString = "This is a string";
console.log(aString instanceof String);     //false
```

The third line returns `false`. We will discuss why this is the case when we discuss prototype chains.

Date objects

JavaScript does not have a date data type. Instead, you can use the **Date** object and its methods to work with dates and times in your applications. A Date object is pretty exhaustive and contains several methods to handle most date- and time-related use cases.

JavaScript treats dates similarly to Java. JavaScript store dates as the number of milliseconds since January 1, 1970, 00:00:00.

You can create a Date object using the following declaration:

```
var dataObject = new Date([parameters]);
```

The parameters for the Date object constructors can be as follows:

- No parameters creates today's date and time. For example, `var today = new Date();`.
- A String representing a date as `Month day, year hours:minutes:seconds`. For example, `var twoThousandFifteen = new Date("December 31, 2015 23:59:59");`. If you omit hours, minutes, or seconds, the value will be set to `0`.
- A set of integer values for the year, month, and day. For example, `var christmas = new Date(2015, 11, 25);`.
- A set of integer values for the year, month, day, hour, minute, and seconds. For example, `var christmas = new Date(2015, 11, 25, 21, 00, 0);`.

Here are some examples on how to create and manipulate dates in JavaScript:

```
var today = new Date();
console.log(today.getDate()); //27
console.log(today.getMonth()); //4
console.log(today.getFullYear()); //2015
console.log(today.getHours()); //23
console.log(today.getMinutes()); //13
console.log(today.getSeconds()); //10
//number of milliseconds since January 1, 1970, 00:00:00 UTC
console.log(today.getTime()); //1432748611392
console.log(today.getTimezoneOffset()); //-330 Minutes

//Calculating elapsed time
var start = Date.now();
// loop for a long time
for (var i=0;i<100000;i++);
var end = Date.now();
var elapsed = end - start; // elapsed time in milliseconds
console.log(elapsed); //71
```

For any serious applications that require fine-grained control over date and time objects, we recommend using libraries such as **Moment.js** (`https://github.com/moment/moment`), **Timezone.js** (`https://github.com/mde/timezone-js`), or **date.js** (`https://github.com/MatthewMueller/date`). These libraries simplify a lot of recurrent tasks for you and help you focus on other important things.

The + operator

The **+** operator, when used as a unary, does not have any effect on a number. However, when applied to a String, the + operator converts it to numbers as follows:

```
var a=25;
a=+a;            //No impact on a's value
console.log(a);  //25

var b="70";
console.log(typeof b); //string
b=+b;           //converts string to number
console.log(b); //70
console.log(typeof b); //number
```

The + operator is used often by a programmer to quickly convert a numeric representation of a String to a number. However, if the String literal is not something that can be converted to a number, you get slightly unpredictable results as follows:

```
var c="foo";
c=+c;           //Converts foo to number
console.log(c);  //NaN
console.log(typeof c);  //number

var zero="";
zero=+zero; //empty strings are converted to 0
console.log(zero);
console.log(typeof zero);
```

We will discuss the effects of the + operator on several other data types later in the text.

The ++ and -- operators

The ++ operator is a shorthand version of adding `1` to a value and -- is a shorthand to subtract `1` from a value. Java and C have equivalent operators and most will be familiar with them. How about this?

```
var a= 1;
var b= a++;
console.log(a); //2
console.log(b); //1
```

Err, what happened here? Shouldn't the `b` variable have the value `2`? The ++ and -- operators are unary operators that can be used either prefix or postfix. The order in which they are used matters. When ++ is used in the prefix position as `++a`, it increments the value before the value is returned from the expression rather than after as with `a++`. Let's see the following code:

```
var a= 1;
var b= ++a;
console.log(a);  //2
console.log(b);  //2
```

Many programmers use the chained assignments to assign a single value to multiple variables as follows:

```
var a, b, c;
a = b = c = 0;
```

This is fine because the assignment operator (=) results in the value being assigned. In this case, `c=0` is evaluated to `0`; this would result in `b=0`, which also evaluates to `0`, and hence, `a=0` is evaluated.

However, a slight change to the previous example yields extraordinary results. Consider this:

```
var a = b = 0;
```

In this case, only the `a` variable is declared with `var`, while the `b` variable is created as an accidental global. (If you are in the strict mode, you will get an error for this.) With JavaScript, be careful what you wish for, you might get it.

Boolean operators

There are three Boolean operators in JavaScript—AND(&), OR(|), and NOT(!).

Before we discuss logical AND and OR operators, we need to understand how they produce a Boolean result. Logical operators are evaluated from left to right and they are tested using the following short-circuit rules:

- **Logical AND**: If the first operand determines the result, the second operand is not evaluated.

  In the following example, I have highlighted the right-hand side expression if it gets executed as part of short-circuit evaluation rules:

  ```
  console.log(true  && true); // true AND true returns true
  console.log(true  && false);// true AND false returns false
  console.log(false && true);// false AND true returns false
  console.log("Foo" && "Bar");// Foo(true) AND Bar(true)
    returns Bar
  console.log(false && "Foo");// false && Foo(true) returns
    false
  console.log("Foo" && false);// Foo(true) && false returns
    false
  console.log(false && (1 == 2));// false && false(1==2) returns
  false
  ```

- **Logical OR**: If the first operand is true, the second operand is not evaluated:

  ```
  console.log(true  || true); // true AND true returns true
  console.log(true  || false);// true AND false returns true
  console.log(false || true);// false AND true returns true
  console.log("Foo" || "Bar");// Foo(true) AND Bar(true) returns Foo
  console.log(false || "Foo");// false && Foo(true) returns Foo
  console.log("Foo" || false);// Foo(true) && false returns Foo
  console.log(false || (1 == 2));// false && false(1==2) returns
  false
  ```

  However, both logical AND and logical OR can also be used for non-Boolean operands. When either the left or right operand is not a primitive Boolean value, AND and OR do not return Boolean values.

Now we will explain the three logical Boolean operators:

- Logical AND(&&): If the first operand object is *falsy*, it returns that object. If its *truthy*, the second operand object is returned:

  ```
  console.log (0 && "Foo");  //First operand is falsy -
    return it
  console.log ("Foo" && "Bar"); //First operand is truthy,
    return the second operand
  ```

- Logical OR(||): If the first operand is *truthy*, it's returned. Otherwise, the second operand is returned:

  ```
  console.log (0 || "Foo");  //First operand is falsy -
    return second operand
  console.log ("Foo" || "Bar"); //First operand is truthy,
    return it
  console.log (0 || false); //First operand is falsy, return
    second operand
  ```

  The typical use of a logical OR is to assign a default value to a variable:

  ```
  function greeting(name){
      name = name || "John";
      console.log("Hello " + name);
  }

  greeting("Johnson"); // alerts "Hi Johnson";
  greeting(); //alerts "Hello John"
  ```

  You will see this pattern frequently in most professional JavaScript libraries. You should understand how the defaulting is done by using a logical OR operator.

- **Logical NOT**: This always returns a Boolean value. The value returned depends on the following:

```
//If the operand is an object, false is returned.
var s = new String("string");
console.log(!s);              //false

//If the operand is the number 0, true is returned.
var t = 0;
console.log(!t);              //true

//If the operand is any number other than 0, false is returned.
var x = 11;
console.log(!x);              //false

//If operand is null or NaN, true is returned
var y =null;
var z = NaN;
console.log(!y);              //true
console.log(!z);              //true
//If operand is undefined, you get true
var foo;
console.log(!foo);            //true
```

Additionally, JavaScript supports C-like ternary operators as follows:

```
var allowedToDrive = (age > 21) ? "yes" : "no";
```

If `(age>21)`, the expression after `?` will be assigned to the `allowedToDrive` variable and the expression after `:` is assigned otherwise. This is equivalent to an if-else conditional statement. Let's see another example:

```
function isAllowedToDrive(age){
  if(age>21){
    return true;
  }else{
    return false;
  }
}
console.log(isAllowedToDrive(22));
```

In this example, the `isAllowedToDrive()` function accepts one integer parameter, `age`. Based on the value of this variable, we return true or false to the calling function. This is a well-known and most familiar if-else conditional logic. Most of the time, if-else keeps the code easier to read. For simpler cases of single conditions, using the ternary operator is also okay, but if you see that you are using the ternary operator for more complicated expressions, try to stick with if-else because it is easier to interpret if-else conditions than a very complex ternary expression.

If-else conditional statements can be nested as follows:

```
if (condition1) {
  statement1
} else if (condition2) {
  statement2
} else if (condition3) {
  statement3
}
..
} else {
  statementN
}
```

Purely as a matter of taste, you can indent the nested `else if` as follows:

```
if (condition1) {
  statement1
} else
    if (condition2) {
```

Do not use assignments in place of a conditional statement. Most of the time, they are used because of a mistake as follows:

```
if(a=b) {
  //do something
}
```

Mostly, this happens by mistake; the intended code was `if(a==b)`, or better, `if(a===b)`. When you make this mistake and replace a conditional statement with an assignment statement, you end up committing a very difficult-to-find bug. However, if you really want to use an assignment statement with an if statement, make sure that you make your intentions very clear.

One way is to put extra parentheses around your assignment statement:

```
if((a=b)){
  //this is really something you want to do
}
```

Another way to handle conditional execution is to use switch-case statements. The switch-case construct in JavaScript is similar to that in C or Java. Let's see the following example:

```
function sayDay(day){
  switch(day){
    case 1: console.log("Sunday");
      break;
    case 2: console.log("Monday");
      break;
    default:
      console.log("We live in a binary world. Go to Pluto");
  }
}

sayDay(1); //Sunday
sayDay(3); //We live in a binary world. Go to Pluto
```

One problem with this structure is that you have `break` out of every case; otherwise, the execution will fall through to the next level. If we remove the `break` statement from the first case statement, the output will be as follows:

```
>sayDay(1);
Sunday
Monday
```

As you can see, if we omit the `break` statement to break the execution immediately after a condition is satisfied, the execution sequence follows to the next level. This can lead to difficult-to-detect problems in your code. However, this is also a popular style of writing conditional logic if you intend to fall through to the next level:

```
function debug(level,msg){
  switch(level){
    case "INFO": //intentional fall-through
    case "WARN" :
    case "DEBUG": console.log(level+ ": " + msg);
      break;
    case "ERROR": console.error(msg);
  }
}
```

```
debug("INFO","Info Message");
debug("DEBUG","Debug Message");
debug("ERROR","Fatal Exception");
```

In this example, we are intentionally letting the execution fall through to write a concise switch-case. If levels are either INFO, WARN, or DEBUG, we use the switch-case to fall through to a single point of execution. We omit the `break` statement for this. If you want to follow this pattern of writing switch statements, make sure that you document your usage for better readability.

Switch statements can have a `default` case to handle any value that cannot be evaluated by any other case.

JavaScript has a while and do-while loop. The while loop lets you iterate a set of expressions till a condition is met. The following first example iterates the statements enclosed within {} till the `i<10` expression is true. Remember that if the value of the `i` counter is already greater than `10`, the loop will not execute at all:

```
var i=0;
while(i<10){
  i=i+1;
  console.log(i);
}
```

The following loop keeps executing till infinity because the condition is always true—this can lead to disastrous effects. Your program can use up all your memory or something equally unpleasant:

```
//infinite loop
while(true){
  //keep doing this
}
```

If you want to make sure that you execute the loop at least once, you can use the do-while loop (sometimes known as a post-condition loop):

```
var choice;
do {
  choice=getChoiceFromUserInput();
} while(!isInputValid(choice));
```

In this example, we are asking the user for an input till we find a valid input from the user. While the user types invalid input, we keep asking for an input to the user. It is always argued that, logically, every do-while loop can be transformed into a while loop. However, a do-while loop has a very valid use case like the one we just saw where you want the condition to be checked only after there has been one execution of the loop block.

JavaScript has a very powerful loop similar to C or Java—the for loop. The for loop is popular because it allows you to define the control conditions of the loop in a single line.

The following example prints `Hello` five times:

```
for (var i=0;i<5;i++){
  console.log("Hello");
}
```

Within the definition of the loop, you defined the initial value of the loop counter `i` to be `0`, you defined the `i<5` exit condition, and finally, you defined the increment factor.

All three expressions in the previous example are optional. You can omit them if required. For example, the following variations are all going to produce the same result as the previous loop:

```
var x=0;
//Omit initialitzation
for (;x<5;x++){
  console.log("Hello");
}

//Omit exit condition
for (var j=0;;j++){
  //exit condition
  if(j>=5){
    break;
  }else{
    console.log("Hello");
  }
}
//Omit increment
for (var k=0; k<5;){
  console.log("Hello");
  k++;
}
```

You can also omit all three of these expressions and write for loops. One interesting idiom used frequently is to use for loops with empty statements. The following loop is used to set all the elements of the array to `100`. Notice how there is no body to the for-loop:

```
var arr = [10, 20, 30];
// Assign all array values to 100
for (i = 0; i < arr.length; arr[i++] = 100);
console.log(arr);
```

The empty statement here is just the single that we see after the for loop statement. The increment factor also modifies the array content. We will discuss arrays later in the book, but here it's sufficient to see that the array elements are set to the `100` value within the loop definition itself.

Equality

JavaScript offers two modes of equality—strict and loose. Essentially, loose equality will perform the type conversion when comparing two values, while strict equality will check the values without any type conversion. A strict equality check is performed by === while a loose equality check is performed by ==.

ECMAScript 6 also offers the `Object.is` method to do a strict equality check like ===. However, `Object.is` has a special handling for NaN: -0 and +0. When *NaN===NaN* and *NaN==NaN* evaluates to false, `Object.is(NaN,NaN)` will return true.

Strict equality using ===

Strict equality compares two values without any implicit type conversions. The following rules apply:

- If the values are of a different type, they are unequal.
- For non-numerical values of the same type, they are equal if their values are the same.
- For primitive numbers, strict equality works for values. If the values are the same, === results in `true`. However, a NaN doesn't equal to any number and `NaN===<a number>` would be a `false`.

Strict equality is always the correct equality check to use. Make it a rule to always use === instead of ==:

Condition	**Output**
`"" === "0"`	false
`0 === ""`	false
`0 === "0"`	false
`false === "false"`	false
`false === "0"`	false
`false === undefined`	false
`false === null`	false
`null === undefined`	false

In case of comparing objects, we get results as follows:

Condition	**Output**
`{} === {};`	false
`new String('bah') === 'bah';`	false
`new Number(1) === 1;`	false
`var bar = {};` `bar === bar;`	true

The following are further examples that you should try on either JS Bin or Node REPL:

```
var n = 0;
var o = new String("0");
var s = "0";
var b = false;

console.log(n === n); // true - same values for numbers
console.log(o === o); // true - non numbers are compared for their
values
console.log(s === s); // true - ditto

console.log(n === o); // false - no implicit type conversion, types
are different
console.log(n === s); // false - types are different
console.log(o === s); // false - types are different
console.log(null === undefined); // false
console.log(o === null); // false
console.log(o === undefined); // false
```

You can use `!==` to handle the **Not Equal To** case while doing strict equality checks.

Weak equality using ==

Nothing should tempt you to use this form of equality. Seriously, stay away from this form. There are many bad things with this form of equality primarily due to the weak typing in JavaScript. The equality operator, ==, first tries to coerce the type before doing a comparison. The following examples show you how this works:

Condition	**Output**
`"" == "0"`	false
`0  == ""`	true
`0  == "0"`	true
`false == "false"`	false
`false == "0"`	true
`false == undefined`	false
`false == null`	false
`null  == undefined`	true

From these examples, it's evident that weak equality can result in unexpected outcomes. Also, implicit type coercion is costly in terms of performance. So, in general, stay away from weak equality in JavaScript.

JavaScript types

We briefly discussed that JavaScript is a dynamic language. If you have a previous experience of strongly typed languages such as Java, you may feel a bit uncomfortable about the complete lack of type checks that you are used to. Purists argue that JavaScript should claim to have **tags** or perhaps **subtypes**, but not types. Though JavaScript does not have the traditional definition of **types**, it is absolutely essential to understand how JavaScript handles data types and coercion internally. Every nontrivial JavaScript program will need to handle value coercion in some form, so it's important that you understand the concept well.

Explicit coercion happens when you modify the type yourself. In the following example, you will convert a number to a String using the `toString()` method and extract the second character out of it:

```
var fortyTwo = 42;
console.log(fortyTwo.toString()[1]); //prints "2"
```

This is an example of an explicit type conversion. Again, we are using the word **type** loosely because type was not enforced anywhere when you declared the `fortyTwo` variable.

However, there are many different ways in which such coercion can happen. Coercion happening explicitly can be easy to understand and mostly reliable; but if you're not careful, coercion can happen in very strange and surprising ways.

Confusion around coercion is perhaps one of the most talked about frustrations for JavaScript developers. To make sure that you never have this confusion in your mind, let's revisit types in JavaScript. We talked about some concepts earlier:

```
typeof 1            === "number";    // true
typeof "1"          === "string";    // true
typeof { age: 39 }  === "object";    // true
typeof Symbol()     === "symbol";    // true
typeof undefined    === "undefined"; // true
typeof true         === "boolean";   // true
```

So far, so good. We already knew this and the examples that we just saw reinforce our ideas about types.

Conversion of a value from one type to another is called **casting** or explicit coercion. JavaScript also does implicit coercion by changing the type of a value based on certain guesses. These guesses make JavaScript work around several cases and unfortunately make it fail quietly and unexpectedly. The following snippet shows cases of explicit and implicit coercion:

```
var t=1;
var u=""+t; //implicit coercion
console.log(typeof t);  //"number"
console.log(typeof u);  //"string"
var v=String(t);  //Explicit coercion
console.log(typeof v);  //"string"
var x=null
console.log(""+x); //"null"
```

It is easy to see what is happening here. When you use `""+t` to a numeric value of `t` (`1`, in this case), JavaScript figures out that you are trying to concatenate *something* with a `""` string. As only strings can be concatenated with other strings, JavaScript goes ahead and converts a numeric `1` to a `"1"` string and concatenates both into a resulting string value. This is what happens when JavaScript is asked to convert values implicitly. However, `String(t)` is a very deliberate call to convert a number to a String. This is an explicit conversion of types. The last bit is surprising. We are concatenating `null` with `""`—shouldn't this fail?

So how does JavaScript do type conversions? How will an abstract value become a String or number or Boolean? JavaScript relies on `toString()`, `toNumber()`, and `toBoolean()` methods to do this internally.

When a non-String value is coerced into a String, JavaScript uses the `toString()` method internally to do this. All primitives have a natural string form—null has a string form of `"null"`, undefined has a string form of `"undefined"`, and so on. For Java developers, this is analogous to a class having a `toString()` method that returns a string representation of the class. We will see exactly how this works in case of objects.

So essentially you can do something similar to the following:

```
var a="abc";
console.log(a.length);
console.log(a.toUpperCase());
```

If you are keenly following and typing all these little snippets, you would have realized something strange in the previous snippet. How are we calling properties and methods on primitives? How come primitives have objects such as properties and methods? They don't.

As we discussed earlier, JavaScript kindly wraps these primitives in their wrappers by default thus making it possible for us to directly access the wrapper's methods and properties as if they were of the primitives themselves.

When any non-number value needs to be coerced into a number, JavaScript uses the `toNumber()` method internally: `true` becomes `1`, `undefined` becomes `NaN`, `false` becomes `0`, and `null` becomes `0`. The `toNumber()` method on strings works with literal conversion and if this fails, the method returns `NaN`.

What about some other cases?

```
typeof null ==="object" //true
```

Well, null is an object? Yes, an especially long-lasting bug makes this possible. Due to this bug, you need to be careful while testing if a value is null:

```
var x = null;
if (!x && typeof x === "object"){
  console.log("100% null");
}
```

What about other things that may have types, such as functions?

```
f = function test() {
  return 12;
}
console.log(typeof f === "function");  //prints "true"
```

What about arrays?

```
console.log (typeof [1,2,3,4]); //"object"
```

Sure enough, they are also objects. We will take a detailed look at functions and arrays later in the book.

In JavaScript, values have types, variables don't. Due to the dynamic nature of the language, variables can hold any value at any time.

JavaScript doesn't does not enforce types, which means that the language doesn't insist that a variable always hold values of the same initial type that it starts out with. A variable can hold a String, and in the next assignment, hold a number, and so on:

```
var a = 1;
typeof a; // "number"
a = false;
typeof a; // "boolean"
```

The `typeof` operator always returns a String:

```
typeof typeof 1; // "string"
```

Automatic semicolon insertion

Although JavaScript is based on the C style syntax, it does not enforce the use of semicolons in the source code.

However, JavaScript is not a semicolon-less language. A JavaScript language parser needs the semicolons in order to understand the source code. Therefore, the JavaScript parser automatically inserts them whenever it encounters a parse error due to a missing semicolon. It's important to note that **automatic semicolon insertion** (**ASI**) will only take effect in the presence of a newline (also known as a line break). Semicolons are not inserted in the middle of a line.

Basically, if the JavaScript parser parses a line where a parser error would occur (a missing expected ;) and it can insert one, it does so. What are the criteria to insert a semicolon? Only if there's nothing but white space and/or comments between the end of some statement and that line's newline/line break.

There have been raging debates on ASI—a feature justifiably considered to be a very bad design choice. There have been epic discussions on the Internet, such as `https://github.com/twbs/bootstrap/issues/3057` and `https://brendaneich.com/2012/04/the-infernal-semicolon/`.

Before you judge the validity of these arguments, you need to understand what is affected by ASI. The following statements are affected by ASI:

- An empty statement
- A var statement
- An expression statement
- A do-while statement
- A continue statement
- A break statement
- A return statement
- A throw statement

The idea behind ASI is to make semicolons optional at the end of a line. This way, ASI helps the parser to determine when a statement ends. Normally, it ends with a semicolon. ASI dictates that a statement also ends in the following cases:

- A line terminator (for example, a newline) is followed by an illegal token
- A closing brace is encountered
- The end of the file has been reached

Let's see the following example:

```
if (a < 1) a = 1 console.log(a)
```

The `console` token is illegal after `1` and triggers ASI as follows:

```
if (a < 1) a = 1; console.log(a);
```

In the following code, the statement inside the braces is not terminated by a semicolon:

```
function add(a,b) { return a+b }
```

ASI creates a syntactically correct version of the preceding code:

```
function add(a,b) { return a+b; }
```

JavaScript style guide

Every programming language develops its own style and structure. Unfortunately, new developers don't put much effort in learning the stylistic nuances of a language. It is very difficult to develop this skill later once you have acquired bad practices. To produce beautiful, readable, and easily maintainable code, it is important to learn the correct style. There are a ton of style suggestions. We will be picking the most practical ones. Whenever applicable, we will discuss the appropriate style. Let's set some stylistic ground rules.

Whitespaces

Though whitespace is not important in JavaScript, the correct use of whitespace can make the code easy to read. The following guidelines will help in managing whitespaces in your code:

- Never mix spaces and tabs.
- Before you write any code, choose between soft indents (spaces) or real tabs. For readability, I always recommend that you set your editor's indent size to two characters—this means two spaces or two spaces representing a real tab.
- Always work with the *show invisibles* setting turned on. The benefits of this practice are as follows:
    - Enforced consistency.
    - Eliminates the end-of-line white spaces.
    - Eliminates blank line white spaces.
    - Commits and diffs that are easier to read.
    - Uses **EditorConfig** (`http://editorconfig.org/`) when possible.

Parentheses, line breaks, and braces

If, else, for, while, and try always have spaces and braces and span multiple lines. This style encourages readability. Let's see the following code:

```
//Cramped style (Bad)
if(condition) doSomeTask();

while(condition) i++;

for(var i=0;i<10;i++) iterate();

//Use whitespace for better readability (Good)
```

```
//Place 1 space before the leading brace.
if (condition) {
  // statements
}

while ( condition ) {
  // statements
}

for ( var i = 0; i < 100; i++ ) {
  // statements
}

// Better:

var i,
    length = 100;

for ( i = 0; i < length; i++ ) {
  // statements
}

// Or...

var i = 0,
    length = 100;

for ( ; i < length; i++ ) {
  // statements
}

var value;

for ( value in object ) {
  // statements
}


if ( true ) {
  // statements
} else {
  // statements
}
```

```
//Set off operators with spaces.
// bad
var x=y+5;

// good
var x = y + 5;

//End files with a single newline character.
// bad
(function(global) {
  // ...stuff...
})(this);

// bad
(function(global) {
  // ...stuff...
})(this);↵
↵

// good
(function(global) {
  // ...stuff...
})(this);↵
```

Quotes

Whether you prefer single or double quotes shouldn't matter; there is no difference in how JavaScript parses them. However, for the sake of consistency, never mix quotes in the same project. Pick one style and stick with it.

End of lines and empty lines

Whitespace can make it impossible to decipher code diffs and changelists. Many editors allow you to automatically remove extra empty lines and end of lines—you should use these.

Type checking

Checking the type of a variable can be done as follows:

```
//String:
typeof variable === "string"
//Number:
typeof variable === "number"
//Boolean:
typeof variable === "boolean"
//Object:
typeof variable === "object"
//null:
variable === null
//null or undefined:
variable == null
```

Type casting

Perform type coercion at the beginning of the statement as follows:

```
// bad
const totalScore = this.reviewScore + '';
// good
const totalScore = String(this.reviewScore);
```

Use `parseInt()` for Numbers and always with a radix for the type casting:

```
const inputValue = '4';
// bad
const val = new Number(inputValue);
// bad
const val = +inputValue;
// bad
const val = inputValue >> 0;
// bad
const val = parseInt(inputValue);
// good
const val = Number(inputValue);
// good
const val = parseInt(inputValue, 10);
```

The following example shows you how to type cast using Booleans:

```
const age = 0;  // bad
const hasAge = new Boolean(age);  // good
const hasAge = Boolean(age); // good
const hasAge = !!age;
```

Conditional evaluation

There are various stylistic guidelines around conditional statements. Let's study the following code:

```
// When evaluating that array has length,
// WRONG:
if ( array.length > 0 ) ...

// evaluate truthiness(GOOD):
if ( array.length ) ...

// When evaluating that an array is empty,
// (BAD):
if ( array.length === 0 ) ...

// evaluate truthiness(GOOD):
if ( !array.length ) ...

// When checking if string is not empty,
// (BAD):
if ( string !== "" ) ...

// evaluate truthiness (GOOD):
if ( string ) ...

// When checking if a string is empty,
// BAD:
if ( string === "" ) ...

// evaluate falsy-ness (GOOD):
if ( !string ) ...

// When checking if a reference is true,
// BAD:
if ( foo === true ) ...
```

```
// GOOD
if ( foo ) ...

// When checking if a reference is false,
// BAD:
if ( foo === false ) ...

// GOOD
if ( !foo ) ...

// this will also match: 0, "", null, undefined, NaN
// If you MUST test for a boolean false, then use
if ( foo === false ) ...

// a reference that might be null or undefined, but NOT false, "" or
0,
// BAD:
if ( foo === null || foo === undefined ) ...

// GOOD
if ( foo == null ) ...

// Don't complicate matters
return x === 0 ? 'sunday' : x === 1 ? 'Monday' : 'Tuesday';

// Better:
if (x === 0) {
    return 'Sunday';
} else if (x === 1) {
    return 'Monday';
} else {
    return 'Tuesday';
}

// Even Better:
switch (x) {
    case 0:
        return 'Sunday';
    case 1:
        return 'Monday';
    default:
        return 'Tuesday';
}
```

Naming

Naming is super important. I am sure that you have encountered code with terse and undecipherable naming. Let's study the following lines of code:

```
//Avoid single letter names. Be descriptive with your naming.
// bad
function q() {

}

// good
function query() {
}

//Use camelCase when naming objects, functions, and instances.
// bad
const OBJEcT = {};
const this_is_object = {};
function c() {}

// good
const thisIsObject = {};
function thisIsFunction() {}

//Use PascalCase when naming constructors or classes.
// bad
function user(options) {
  this.name = options.name;
}

const bad = new user({
  name: 'nope',
});

// good
class User {
  constructor(options) {
    this.name = options.name;
  }
}
```

```
const good = new User({
  name: 'yup',
});

// Use a leading underscore _ when naming private properties.
// bad
this.__firstName__ = 'Panda';
this.firstName_ = 'Panda';

// good
this._firstName = 'Panda';
```

The eval() method is evil

The `eval()` method, which takes a String containing JavaScript code, compiles it and runs it, is one of the most misused methods in JavaScript. There are a few situations where you will find yourself using `eval()`, for example, when you are building an expression based on the user input.

However, most of the time, `eval()` is used is just because it gets the job done. The `eval()` method is too hacky and makes the code unpredictable. It's slow, unwieldy, and tends to magnify the damage when you make a mistake. If you are considering using `eval()`, then there is probably a better way.

The following snippet shows the usage of `eval()`:

```
console.log(typeof eval(new String("1+1"))); // "object"
console.log(eval(new String("1+1")));        //1+1
console.log(eval("1+1"));                    // 2
console.log(typeof eval("1+1"));             // returns "number"
var expression = new String("1+1");
console.log(eval(expression.toString()));    //2
```

I will refrain from showing other uses of `eval()` and make sure that you are discouraged enough to stay away from it.

The strict mode

ECMAScript 5 has a strict mode that results in cleaner JavaScript, with fewer unsafe features, more warnings, and more logical behavior. The normal (non-strict) mode is also called **sloppy mode**. The strict mode can help you avoid a few sloppy programming practices. If you are starting a new JavaScript project, I would highly recommend that you use the strict mode by default.

You switch on the strict mode by typing the following line first in your JavaScript file or in your `<script>` element:

```
'use strict';
```

Note that JavaScript engines that don't support ECMAScript 5 will simply ignore the preceding statement and continue as non-strict mode.

If you want to switch on the strict mode per function, you can do it as follows:

```
function foo() {
    'use strict';

}
```

This is handy when you are working with a legacy code base where switching on the strict mode everywhere may break things.

If you are working on an existing legacy code, be careful because using the strict mode can break things. There are caveats on this:

Enabling the strict mode for an existing code can break it

The code may rely on a feature that is not available anymore or on behavior that is different in a sloppy mode than in a strict mode. Don't forget that you have the option to add single strict mode functions to files that are in the sloppy mode.

Package with care

When you concatenate and/or minify files, you have to be careful that the strict mode isn't switched off where it should be switched on or vice versa. Both can break code.

The following sections explain the strict mode features in more detail. You normally don't need to know them as you will mostly get warnings for things that you shouldn't do anyway.

Variables must be declared in strict mode

All variables must be explicitly declared in strict mode. This helps to prevent typos. In the sloppy mode, assigning to an undeclared variable creates a global variable:

```
function sloppyFunc() {
  sloppyVar = 123;
} sloppyFunc();  // creates global variable `sloppyVar`
console.log(sloppyVar);  // 123
```

In the strict mode, assigning to an undeclared variable throws an exception:

```
function strictFunc() {
  'use strict';
  strictVar = 123;
}
strictFunc();  // ReferenceError: strictVar is not defined
```

The eval() function is cleaner in strict mode

In strict mode, the `eval()` function becomes less quirky: variables declared in the evaluated string are not added to the scope surrounding `eval()` anymore.

Features that are blocked in strict mode

The with statement is not allowed. (We will discuss this in the book later.) You get a syntax error at compile time (when loading the code).

In the sloppy mode, an integer with a leading zero is interpreted as octal (base 8) as follows:

```
> 010 === 8 true
```

In strict mode, you get a syntax error if you use this kind of literal:

```
function f() {
'use strict';
return 010
}
//SyntaxError: Octal literals are not allowed in
```

Running JSHint

JSHint is a program that flags suspicious usage in programs written in JavaScript. The core project consists of a library itself as well as a **command line interface** (**CLI**) program distributed as a Node module.

If you have Node.js installed, you can install JSHint using `npm` as follows:

```
npm install jshint -g
```

Once JSHint is installed, you can lint a single or multiple JavaScript files. Save the following JavaScript code snippet in the `test.js` file:

```
function f(condition) {
  switch (condition) {
  case 1:
    console.log(1);
  case 2:
    console.log(1);
  }
}
```

When we run the file using JSHint, it will warn us of a missing `break` statement in the switch case as follows:

```
>jshint test.js
test.js: line 4, col 19, Expected a 'break' statement before 'case'.
1 error
```

JSHint is configurable to suit your needs. Check the documentation at `http://jshint.com/docs/` to see how you can customize JSHint according to your project needs. I use JSHint extensively and suggest you start using it. You will be surprised to see how many hidden bugs and stylistic issues you will be able to fix in your code with such a simple tool.

You can run JSHint at the root of your project and lint the entire project. You can place JSHint directives in the `.jshintrc` file. This file may look something as follows:

```
{
     "asi": false,
     "expr": true,
     "loopfunc": true,
     "curly": false,
     "evil": true,
     "white": true,
     "undef": true,
     "indent": 4
}
```

Summary

In this chapter, we set some foundations around JavaScript grammar, types, and stylistic considerations. We have consciously not talked about other important aspects such as functions, variable scopes, and closures primarily because they deserve their own place in this book. I am sure that this chapter helps you understand some of the primary concepts of JavaScript. With these foundations in place, we will take a look at how we can write professional quality JavaScript code.

2 Functions, Closures, and Modules

In the previous chapter, we deliberately did not discuss certain aspects of JavaScript. These are some of the features of the language that give JavaScript its power and elegance. If you are an intermediate- or advanced-level JavaScript programmer, you may be actively using objects and functions. In many cases, however, developers stumble at these fundamental levels and develop a half-baked or sometimes wrong understanding of the core JavaScript constructs. There is generally a very poor understanding of the concept of closures in JavaScript, due to which many programmers cannot use the functional aspects of JavaScript very well. In JavaScript, there is a strong interconnection between objects, functions, and closures. Understanding the strong relationship between these three concepts can vastly improve our JavaScript programming ability, giving us a strong foundation for any type of application development.

Functions are fundamental to JavaScript. Understanding functions in JavaScript is the single most important weapon in your arsenal. The most important fact about functions is that in JavaScript, functions are first-class objects. They are treated like any other JavaScript object. Just like other JavaScript data types, they can be referenced by variables, declared with literals, and even passed as function parameters.

As with any other object in JavaScript, functions have the following capabilities:

- They can be created via literals
- They can be assigned to variables, array entries, and properties of other objects
- They can be passed as arguments to functions
- They can be returned as values from functions
- They can possess properties that can be dynamically created and assigned

We will talk about each of these unique abilities of a JavaScript function in this chapter and the rest of the book.

A function literal

One of the most important concepts in JavaScript is that the functions are the primary unit of execution. Functions are the pieces where you will wrap all your code, hence they will give your programs a structure.

JavaScript functions are declared using a function literal.

Function literals are composed of the following four parts:

- The function keyword.
- An optional name that, if specified, must be a valid JavaScript identifier.
- A list of parameter names enclosed in parentheses. If there are no parameters to the function, you need to provide empty parentheses.
- The body of the function as a series of JavaScript statements enclosed in braces.

A function declaration

The following is a very trivial example to demonstrate all the components of a function declaration:

```
function add(a,b){
  return a+b;
}
c = add(1,2);
console.log(c);  //prints 3
```

The declaration begins with a `function` keyword followed by the function name. The function name is optional. If a function is not given a name, it is said to be anonymous. We will see how anonymous functions are used. The third part is the set of parameters of the function, wrapped in parentheses. Within the parentheses is a set of zero or more parameter names separated by commas. These names will be defined as variables in the function, and instead of being initialized to undefined, they will be initialized to the arguments supplied when the function is invoked. The fourth part is a set of statements wrapped in curly braces. These statements are the body of the function. They are executed when the function is invoked.

This method of function declaration is also known as **function statement**. When you declare functions like this, the content of the function is compiled and an object with the same name as the function is created.

Another way of function declaration is via **function expressions**:

```
var add = function(a,b){
  return a+b;
}
c = add(1,2);
console.log(c);  //prints 3
```

Here, we are creating an anonymous function and assigning it to an `add` variable; this variable is used to invoke the function as in the earlier example. One problem with this style of function declaration is that we cannot have recursive calls to this kind of function. Recursion is an elegant style of coding where the function calls itself. You can use named function expressions to solve this limitation. As an example, refer to the following function to compute the factorial of a given number, `n`:

```
var facto = function factorial(n) {
  if (n <= 1)
    return 1;
  return n * factorial(n - 1);
};
console.log(facto(3));  //prints 6
```

Here, instead of creating an anonymous function, you are creating a named function. Now, because the function has a name, it can call itself recursively.

Finally, you can create self-invoking function expressions (we will discuss them later):

```
(function sayHello() {
  console.log("hello!");
})();
```

Once defined, a function can be called in other JavaScript functions. After the function body is executed, the caller code (that executed the function) continues to execute. You can also pass a function as a parameter to another function:

```
function changeCase(val) {
  return val.toUpperCase();
}
function demofunc(a, passfunction) {
  console.log(passfunction(a));
}
demofunc("smallcase", changeCase);
```

In the preceding example, we are calling the `demofunc()` function with two parameters. The first parameter is the string that we want to convert to uppercase and the second one is the function reference to the `changeCase()` function. In `demofunc()`, we call the `changeCase()` function via its reference passed to the `passfunction` argument. Here we are passing a function reference as an argument to another function. This powerful concept will be discussed in detail later in the book when we discuss callbacks.

A function may or may not return a value. In the previous examples, we saw that the `add` function returned a value to the calling code. Apart from returning a value at the end of the function, calling `return` explicitly allows you to conditionally return from a function:

```
var looper = function(x){
  if (x%5===0) {
    return;
  }
  console.log(x)
}
for(var i=1;i<10;i++){
  looper(i);
}
```

This code snippet prints `1`, `2`, `3`, `4`, `6`, `7`, `8`, and `9`, and not 5. When the `if (x%5===0)` condition is evaluated to true, the code simply returns from the function and the rest of the code is not executed.

Functions as data

In JavaScript, functions can be assigned to variables, and variables are data. You will shortly see that this is a powerful concept. Let's see the following example:

```
var say = console.log;
say("I can also say things");
```

In the preceding example, we assigned the familiar `console.log()` function to the say variable. Any function can be assigned to a variable as shown in the preceding example. Adding parentheses to the variable will invoke it. Moreover, you can pass functions in other functions as parameters. Study the following example carefully and type it in JS Bin:

```
var validateDataForAge = function(data) {
  person = data();
  console.log(person);
  if (person.age <1 || person.age > 99){
    return true;
  }else{
    return false;
  }
};

var errorHandlerForAge = function(error) {
  console.log("Error while processing age");
};

function parseRequest(data,validateData,errorHandler) {
  var error = validateData(data);
  if (!error) {
    console.log("no errors");
  } else {
    errorHandler();
  }
}

var generateDataForScientist = function() {
  return {
    name: "Albert Einstein",
    age : Math.floor(Math.random() * (100 - 1)) + 1,
  };
};
```

```
var generateDataForComposer = function() {
  return {
    name: "J S Bach",
    age : Math.floor(Math.random() * (100 - 1)) + 1,
  };
};

//parse request
parseRequest(generateDataForScientist, validateDataForAge,
errorHandlerForAge);
parseRequest(generateDataForComposer, validateDataForAge,
errorHandlerForAge);
```

In this example, we are passing functions as parameters to a `parseRequest()` function. We are passing different functions for two different calls, `generateDataForScientist` and `generateDataForComposers`, while the other two functions remain the same. You can observe that we defined a generic `parseRequest()`. It takes three functions as arguments, which are responsible for stitching together the specifics: the data, validator, and error handler. The `parseRequest()` function is fully extensible and customizable, and because it will be invoked by every request, there is a single, clean debugging point. I am sure that you have started to appreciate the incredible power that JavaScript functions provide.

Scoping

For beginners, JavaScript scoping is slightly confusing. These concepts may seem straightforward; however, they are not. Some important subtleties exist that must be understood in order to master the concept. So what is Scope? In JavaScript, scope refers to the current context of code.

A variable's scope is the context in which the variable exists. The scope specifies from where you can access a variable and whether you have access to the variable in that context. Scopes can be globally or locally defined.

Global scope

Any variable that you declare is by default defined in global scope. This is one of the most annoying language design decisions taken in JavaScript. As a global variable is visible in all other scopes, a global variable can be modified by any scope. Global variables make it harder to run loosely coupled subprograms in the same program/module. If the subprograms happen to have global variables that share the same names, then they will interfere with each other and likely fail, usually in difficult-to-diagnose ways. This is sometimes known as namespace clash. We discussed global scope in the previous chapter but let's revisit it briefly to understand how best to avoid this.

You can create a global variable in two ways:

- The first way is to place a var statement outside any function. Essentially, any variable declared outside a function is defined in the global scope.
- The second way is to omit the var statement while declaring a variable (also called implied globals). I think this was designed as a convenience for new programmers but turned out to be a nightmare. Even within a function scope, if you omit the var statement while declaring a variable, it's created by default in the global scope. This is nasty. You should always run your program against **ESLint** or **JSHint** to let them flag such violations. The following example shows how global scope behaves:

  ```
  //Global Scope
  var a = 1;
  function scopeTest() {
    console.log(a);
  }
  scopeTest();  //prints 1
  ```

Here we are declaring a variable outside the function and in the global scope. This variable is available in the `scopeTest()` function. If you assign a new value to a global scope variable within a function scope (local), the original value in the global scope is overwritten:

```
//Global Scope
var a = 1;
function scopeTest() {
  a = 2; //Overwrites global variable 2, you omit 'var'
  console.log(a);
}
console.log(a); //prints 1
scopeTest();  //prints 2
console.log(a); //prints 2 (global value is overwritten)
```

Local scope

Unlike most programming languages, JavaScript does not have block-level scope (variables scoped to surrounding curly brackets); instead, JavaScript has function-level scope. Variables declared within a function are local variables and are only accessible within that function or by functions inside that function:

```
var scope_name = "Global";
function showScopeName () {
  // local variable; only accessible in this function
  var scope_name = "Local";
  console.log (scope_name); // Local
}
console.log (scope_name);     //prints - Global
showScopeName();             //prints – Local
```

Function-level scope versus block-level scope

JavaScript variables are scoped at the function level. You can think of this as a small bubble getting created that prevents the variable to be visible from outside this bubble. A function creates such a bubble for variables declared inside the function. You can visualize the bubbles as follows:

```
-GLOBAL SCOPE---------------------------------------------|
var g =0;                                                 |
function foo(a) { -----------------------|                |
    var b = 1;                           |                |
    //code                               |                |
    function bar() { ------|             |                |
        // ...             |ScopeBar     | ScopeFoo       |
    }               ------|              |                |
    // code                              |                |
    var c = 2;                           |                |
}----------------------------------------|                |
foo();   //WORKS                                          |
bar();   //FAILS                                          |
----------------------------------------------------------|
```

JavaScript uses scope chains to establish the scope for a given function. There is typically one global scope, and each function defined has its own nested scope. Any function defined within another function has a local scope that is linked to the outer function. *It's always the position in the source that defines the scope*. When resolving a variable, JavaScript starts at the innermost scope and searches outwards. With this, let's look at various scoping rules in JavaScript.

In the preceding crudely drawn visual, you can see that the `foo()` function is defined in the global scope. The `foo()` function has its local scope and access to the `g` variable because it's in the global scope. The `a`, `b`, and `c` variables are available in the local scope because they are defined within the function scope. The `bar()` function is also declared within the function scope and is available within the `foo()` function. However, once the function scope is over, the `bar()` function is not available. You cannot see or call the `bar()` function from outside the `foo()` function—a scope bubble.

Now that the `bar()` function also has its own function scope (bubble), what is available in here? The `bar()` function has access to the `foo()` function and all the variables created in the parent scope of the `foo()` function—`a`, `b`, and `c`. The `bar()` function also has access to the global scoped variable, `g`.

This is a powerful idea. Take a moment to think about it. We just discussed how rampant and uncontrolled global scope can get in JavaScript. How about we take an arbitrary piece of code and wrap it around with a function? We will be able to hide and create a scope bubble around this piece of code. Creating the correct scope using function wrapping will help us create correct code and prevent difficult-to-detect bugs.

Another advantage of the function scope and hiding variables and functions within this scope is that you can avoid collisions between two identifiers. The following example shows such a bad case:

```
function foo() {
  function bar(a) {
    i = 2; // changing the 'i' in the enclosing scope's for-loop
    console.log(a+i);
  }
  for (var i=0; i<10; i++) {
    bar(i); // infinite loop
  }
}
foo();
```

In the `bar()` function, we are inadvertently modifying the value of `i=2`. When we call `bar()` from within the `for` loop, the value of the `i` variable is set to `2` and we never come out of an infinite loop. This is a bad case of namespace collision.

So far, using functions as a scope sounds like a great way to achieve modularity and correctness in JavaScript. Well, though this technique works, it's not really ideal. The first problem is that we must create a named function. If we keep creating such functions just to introduce the function scope, we pollute the global scope or parent scope. Additionally, we have to keep calling such functions. This introduces a lot of boilerplate, which makes the code unreadable over time:

```
var a = 1;
//Lets introduce a function -scope
//1. Add a named function foo() into the global scope
function foo() {
  var a = 2;
  console.log( a ); // 2
}
//2. Now call the named function foo()
foo();
console.log( a ); // 1
```

We introduced the function scope by creating a new function `foo()` to the global scope and called this function later to execute the code.

In JavaScript, you can solve both these problems by creating functions that immediately get executed. Carefully study and type the following example:

```
var a = 1;
//Lets introduce a function -scope
//1. Add a named function foo() into the global scope
(function foo() {
    var a = 2;
    console.log( a ); // 2
})(); //<---this function executes immediately
console.log( a ); // 1
```

Notice that the wrapping function statement starts with `function`. This means that instead of treating the function as a standard declaration, the function is treated as a function expression.

The `(function foo(){      })` statement as an expression means that the identifier `foo` is found only in the scope of the `foo()` function, not in the outer scope. Hiding the name `foo` in itself means that it does not pollute the enclosing scope unnecessarily. This is so useful and far better. We add `()` after the function expression to execute it immediately. So the complete pattern looks as follows:

```
(function foo(){ /* code */ })();
```

This pattern is so common that it has a name: **IIFE**, which stands for **Immediately Invoked Function Expression**. Several programmers omit the function name when they use IIFE. As the primary use of IIFE is to introduce function-level scope, naming the function is not really required. We can write the earlier example as follows:

```
var a = 1;
(function() {
    var a = 2;
    console.log( a ); // 2
})();
console.log( a ); // 1
```

Here we are creating an anonymous function as IIFE. While this is identical to the earlier named IIFE, there are a few drawbacks of using anonymous IIFEs:

- As you can't see the function name in the stack traces, debugging such code is very difficult
- You cannot use recursion on anonymous functions (as we discussed earlier)
- Overusing anonymous IIFEs sometimes results in unreadable code

Douglas Crockford and a few other experts recommend a slight variation of IIFE:

```
(function(){ /* code */ }());
```

Both these IIFE forms are popular and you will see a lot of code using both these variations.

You can pass parameters to IIFEs. The following example shows you how to pass parameters to IIFEs:

```
(function foo(b) {
    var a = 2;
    console.log( a + b );
})(3); //prints 5
```

Inline function expressions

There is another popular usage of inline function expressions where the functions are passed as parameters to other functions:

```
function setActiveTab(activeTabHandler, tab){
  //set active tab
  //call handler
  activeTabHandler();
}
setActiveTab( function (){
  console.log( "Setting active tab" );
}, 1 );
//prints "Setting active tab"
```

Again, you can name this inline function expression to make sure that you get a correct stack trace while you are debugging the code.

Block scopes

As we discussed earlier, JavaScript does not have the concept of block scopes. Programmers familiar with other languages such as Java or C find this very uncomfortable. **ECMAScript 6** (**ES6**) introduces the **let** keyword to introduce traditional block scope. This is so incredibly convenient that if you are sure your environment is going to support ES6, you should always use the `let` keyword. See the following code:

```
var foo = true;
if (foo) {
  let bar = 42; //variable bar is local in this block { }
  console.log( bar );
}
console.log( bar ); // ReferenceError
```

However, as things stand today, ES6 is not supported by default in most popular browsers.

This chapter so far should have given you a fair understanding of how scoping works in JavaScript. If you are still unclear, I would suggest that you stop here and revisit the earlier sections of this chapter. Research your doubts on the Internet or put your questions on Stack Overflow. In short, make sure that you have no doubts related to the scoping rules.

It is very natural for us to think of code execution happening from top to bottom, line by line. This is how most of JavaScript code is executed but with some exceptions.

Consider the following code:

```
console.log( a );
var a = 1;
```

If you said this is an invalid code and will result in `undefined` when we call `console.log()`, you are absolutely correct. However, what about this?

```
a = 1;
var a;
console.log( a );
```

What should be the output of the preceding code? It is natural to expect `undefined` as the `var a` statement comes after `a = 1`, and it would seem natural to assume that the variable is redefined and thus assigned the default `undefined`. However, the output will be `1`.

When you see `var a = 1`, JavaScript splits it into two statements: `var a` and `a = 1`. The first statement, the declaration, is processed during the compilation phase. The second statement, the assignment, is left in place for the execution phase.

So the preceding snippet would actually be executed as follows:

```
var a;   //----Compilation phase

a = 1;    //------execution phase
console.log( a );
```

The first snippet is actually executed as follows:

```
var a;     //-----Compilation phase

console.log( a );
a = 1;     //------execution phase
```

So, as we can see, variable and function declarations are moved up to the top of the code during compilation phase—this is also popularly known as **hoisting**. It is very important to remember that only the declarations themselves are hoisted, while any assignments or other executable logic are left in place. The following snippet shows you how function declarations are hoisted:

```
foo();
function foo() {
  console.log(a); // undefined
  var a = 1;
}
```

The declaration of the `foo()` function is hoisted such that we are able to execute the function before defining it. One important aspect of hoisting is that it works per scope. Within the `foo()` function, declaration of the a variable will be hoisted to the top of the `foo()` function, and not to the top of the program. The actual execution of the `foo()` function with hoisting will be something as follows:

```
function foo() {
  var a;
  console.log(a); // undefined
  a = 1;
}
```

We saw that function declarations are hoisted but function expressions are not. The next section explains this case.

Function declarations versus function expressions

We saw two ways by which functions are defined. Though they both serve identical purposes, there is a difference between these two types of declarations. Check the following example:

```
//Function expression
functionOne();
//Error
//"TypeError: functionOne is not a function

var functionOne = function() {
  console.log("functionOne");
};
//Function declaration
functionTwo();
//No error
//Prints - functionTwo

function functionTwo() {
  console.log("functionTwo");
}
```

A function declaration is processed when execution enters the context in which it appears before any step-by-step code is executed. The function that it creates is given a proper name (`functionTwo()` in the preceding example) and this name is put in the scope in which the declaration appears. As it's processed before any step-by-step code in the same context, calling `functionTwo()` before defining it works without an error.

However, `functionOne()` is an anonymous function expression, evaluated when it's reached in the step-by-step execution of the code (also called runtime execution); we have to declare it before we can invoke it.

So essentially, the function declaration of `functionTwo()` was hoisted while the function expression of `functionOne()` was executed when line-by-line execution encountered it.

Both function declarations and variable declarations are hoisted but functions are hoisted first, and then variables.

One thing to remember is that you should never use function declarations conditionally. This behavior is non-standardized and can behave differently across platforms. The following example shows such a snippet where we try to use function declarations conditionally. We are trying to assign different function body to function `sayMoo()` but such a conditional code is not guaranteed to work across all browsers and can result in unpredictable results:

```
// Never do this - different browsers will behave differently
if (true) {
  function sayMoo() {
    return 'trueMoo';
  }
}
else {
  function sayMoo() {
    return 'falseMoo';
  }
}
foo();
```

However, it's perfectly safe and, in fact, smart to do the same with function expressions:

```
var sayMoo;
if (true) {
  sayMoo = function() {
    return 'trueMoo';
  };
}
else {
  sayMoo = function() {
    return 'falseMoo';
  };
}
foo();
```

If you are curious to know why you should not use function declarations in conditional blocks, read on; otherwise, you can skip the following paragraph.

Function declarations are allowed to appear only in the program or function body. They cannot appear in a block (`{ ... }`). Blocks can only contain statements and not function declarations. Due to this, almost all implementations of JavaScript have behavior different from this. It is always advisable to *never* use function declarations in a conditional block.

Function expressions, on the other hand, are very popular. A very common pattern among JavaScript programmers is to fork function definitions based on some kind of a condition. As such forks usually happen in the same scope, it is almost always necessary to use function expressions.

The arguments parameter

The arguments parameter is a collection of all the arguments passed to the function. The collection has a property named `length` that contains the count of arguments, and the individual argument values can be obtained using an array indexing notation. Okay, we lied a bit. The arguments parameter is not a JavaScript array, and if you try to use array methods on arguments, you'll fail miserably. You can think of arguments as an array-like structure. This makes it possible to write functions that take an unspecified number of parameters. The following snippet shows you how you can pass a variable number of arguments to the function and iterate through them using an arguments array:

```
var sum = function () {
  var i, total = 0;
  for (i = 0; i < arguments.length; i += 1) {
    total += arguments[i];
  }
  return total;
};
console.log(sum(1,2,3,4,5,6,7,8,9)); // prints 45
console.log(sum(1,2,3,4,5)); // prints 15
```

As we discussed, the arguments parameter is not really an array; it is possible to convert it to an array as follows:

```
var args = Array.prototype.slice.call(arguments);
```

Once converted to an array, you can manipulate the list as you wish.

The this parameter

Whenever a function is invoked, in addition to the parameters that represent the explicit arguments that were provided on the function call, an implicit parameter named `this` is also passed to the function. It refers to an object that's implicitly associated with the function invocation, termed as a **function context**. If you have coded in Java, the `this` keyword will be familiar to you; like Java, `this` points to an instance of the class in which the method is defined.

Equipped with this knowledge, let's talk about various invocation methods.

Invocation as a function

If a function is not invoked as a method, constructor, or via `apply()` or `call()`, it's simply invoked *as a function*:

```
function add() {}
add();
var substract = function() {

};
substract();
```

When a function is invoked with this pattern, `this` is bound to the global object. Many experts believe this to be a bad design choice. It is natural to assume that `this` would be bound to the parent context. When you are in a situation such as this, you can capture the value of `this` in another variable. We will focus on this pattern later.

Invocation as a method

A method is a function tied to a property on an object. For methods, `this` is bound to the object on invocation:

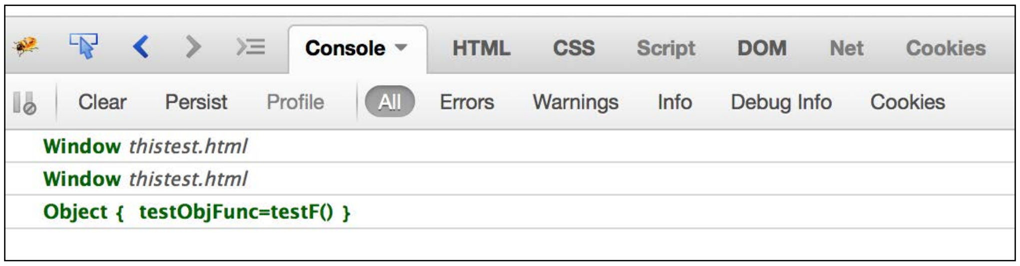
```
var person = {
  name: 'Albert Einstein',
  age: 66,
  greet: function () {
    console.log(this.name);
  }
};
person.greet();
```

In this example, `this` is bound to the person object on invoking `greet` because `greet` is a method of person. Let's see how this behaves in both these invocation patterns.

Let's prepare this HTML and JavaScript harness:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>This test</title>
  <script type="text/javascript">
    function testF(){ return this; }
    console.log(testF());
    var testFCopy = testF;
    console.log(testFCopy());
    var testObj = {
      testObjFunc: testF
    };
    console.log(testObj.testObjFunc ());
  </script>
</head>
<body>
</body>
</html>
```

In the **Firebug** console, you can see the following output:



```
Window thistest.html
Window thistest.html
Object { testObjFunc=testF() }
```

The first two method invocations were invocation as a function; hence, the `this` parameter pointed to the global context (`Window`, in this case).

Next, we define an object with a `testObj` variable with a property named `testObjFunc` that receives a reference to `testF()`—don't fret if you are not really aware of object creation yet. By doing this, we created a `testObjMethod()` method. Now, when we invoke this method, we expect the function context to be displayed when we display the value of `this`.

Invocation as a constructor

Constructor functions are declared just like any other functions and there's nothing special about a function that's going to be used as a constructor. However, the way in which they are invoked is very different.

To invoke the function as a constructor, we precede the function invocation with the **new** keyword. When this happens, `this` is bound to the new object.

Before we discuss more, let's take a quick introduction to object orientation in JavaScript. We will, of course, discuss the topic in great detail in the next chapter. JavaScript is a prototypal inheritance language. This means that objects can inherit properties directly from other objects. The language is class-free. Functions that are designed to be called with the `new` prefix are called constructors. Usually, they are named using **PascalCase** as opposed to **CamelCase** for easier distinction. In the following example, notice that the `greet` function uses this to access the `name` property. The `this` parameter is bound to `Person`:

```
var Person = function (name) {
  this.name = name;
};
Person.prototype.greet = function () {
  return this.name;
};
var albert = new Person('Albert Einstein');
console.log(albert.greet());
```

We will discuss this particular invocation method when we study objects in the next chapter.

Invocation using apply() and call() methods

We said earlier that JavaScript functions are objects. Like other objects, they also have certain methods. To invoke a function using its `apply()` method, we pass two parameters to `apply()`: the object to be used as the function context and an array of values to be used as the invocation arguments. The `call()` method is used in a similar manner, except that the arguments are passed directly in the argument list rather than as an array.

Anonymous functions

We introduced you to anonymous functions a bit earlier in this chapter, and as they're a crucial concept, we will take a detailed look at them. For a language inspired by Scheme, anonymous functions are an important logical and structural construct.

Anonymous functions are typically used in cases where the function doesn't need to have a name for later reference. Let's look at some of the most popular usages of anonymous functions.

Anonymous functions while creating an object

An anonymous function can be assigned to an object property. When we do that, we can call that function with a dot (`.`) operator. If you are coming from a Java or other OO language background, you will find this very familiar. In such languages, a function, which is part of a class is generally called with a notation—`Class.function()`. Let's consider the following example:

```
var santa = {
  say :function(){
    console.log("ho ho ho");
  }
}
santa.say();
```

In this example, we are creating an object with a `say` property, which is an anonymous function. In this particular case, this property is known as a method and not a function. We don't need to name this function because we are going to invoke it as the object property. This is a popular pattern and should come in handy.

Anonymous functions while creating a list

Here, we are creating two anonymous functions and adding them to an array. (We will take a detailed look at arrays later.) Then, you loop through this array and execute the functions in a loop:

```
<script type="text/javascript">
var things = [
  function() { alert("ThingOne") },
  function() { alert("ThingTwo") },
];
for(var x=0; x<things.length; x++) {
  things[x]();
}
</script>
```

Anonymous functions as a parameter to another function

This is one of the most popular patterns and you will find such code in most professional libraries:

```
// function statement
function eventHandler(event){
  event();
}

eventHandler(function(){
  //do a lot of event related things
  console.log("Event fired");
});
```

You are passing the anonymous function to another function. In the receiving function, you are executing the function passed as a parameter. This can be very convenient if you are creating single-use functions such as object methods or event handlers. The anonymous function syntax is more concise than declaring a function and then doing something with it as two separate steps.

Anonymous functions in conditional logic

You can use anonymous function expressions to conditionally change behavior. The following example shows this pattern:

```
var shape;
if(shape_name === "SQUARE") {
  shape = function() {
    return "drawing square";
  }
}
else {
  shape = function() {
    return "drawing square";
  }
}
alert(shape());
```

Here, based on a condition, we are assigning a different implementation to the `shape` variable. This pattern can be very useful if used with care. Overusing this can result in unreadable and difficult-to-debug code.

Later in this book, we will look at several functional tricks such as **memoization** and caching function calls. If you have reached here by quickly reading through the entire chapter, I would suggest that you stop for a while and contemplate on what we have discussed so far. The last few pages contain a ton of information and it will take some time for all this information to sink in. I would suggest that you reread this chapter before proceeding further. The next section will focus on closures and the module pattern.

Closures

Traditionally, closures have been a feature of purely functional programming languages. JavaScript shows its affinity with such functional programming languages by considering closures integral to the core language constructs. Closures are gaining popularity in mainstream JavaScript libraries and advanced production code because they let you simplify complex operations. You will hear experienced JavaScript programmers talking almost reverently about closures—as if they are some magical construct far beyond the reach of the intellect that common men possess. However, this is not so. When you study this concept, you will find closures to be very obvious, almost matter-of-fact. Till you reach closure enlightenment, I suggest you read and reread this chapter, research on the Internet, write code, and read JavaScript libraries to understand how closures behave—but do not give up.

The first realization that you must have is that closure is everywhere in JavaScript. It is not a hidden special part of the language.

Before we jump into the nitty-gritty, let's quickly refresh the lexical scope in JavaScript. We discussed in great detail how lexical scope is determined at the function level in JavaScript. Lexical scope essentially determines where and how all identifiers are declared and predicts how they will be looked up during execution.

In a nutshell, closure is the scope created when a function is declared that allows the function to access and manipulate variables that are external to this function. In other words, closures allow a function to access all the variables, as well as other functions, that are in scope when the function itself is declared.

Let's look at some example code to understand this definition:

```
var outer = 'I am outer'; //Define a value in global scope
function outerFn() { //Declare a a function in global scope
  console.log(outer);
}
outerFn(); //prints - I am outer
```

Were you expecting something shiny? No, this is really the most ordinary case of a closure. We are declaring a variable in the global scope and declaring a function in the global scope. In the function, we are able to access the variable declared in the global scope—`outer`. So essentially, the outer scope for the `outerFn()` function is a closure and always available to `outerFn()`. This is a good start but perhaps then you are not sure why this is such a great thing.

Let's make things a bit more complex:

```
var outer = 'Outer'; //Variable declared in global scope
var copy;
function outerFn(){  //Function declared in global scope

  var inner = 'Inner'; //Variable has function scope only, can not be
  //accessed from outside

  function innerFn(){     //Inner function within Outer function,
    //both global context and outer
    //context are available hence can access
    //'outer' and 'inner'
    console.log(outer);
    console.log(inner);
  }
  copy=innerFn;          //Store reference to inner function,
  //because 'copy' itself is declared
  //in global context, it will be available
  //outside also
}
outerFn();
copy();  //Cant invoke innerFn() directly but can invoke via a
//variable declared in global scope
```

Let's analyze the preceding example. In `innerFn()`, the `outer` variable is available as it's part of the global context. We're executing the inner function after the outer function has been executed via copying a reference to the function to a global reference variable, `copy`. When `innerFn()` executes, the scope in `outerFn()` is gone and not visible at the point at which we're invoking the function through the copy variable. So shouldn't the following line fail?

```
console.log(inner);
```

Should the `inner` variable be undefined? However, the output of the preceding code snippet is as follows:

```
"Outer"
"Inner"
```

What phenomenon allows the `inner` variable to still be available when we execute the inner function, long after the scope in which it was created has gone away? When we declared `innerFn()` in `outerFn()`, not only was the function declaration defined, but a closure was also created that encompasses not only the function declaration, but also all the variables that are in scope at the point of the declaration. When `innerFn()` executes, even if it's executed after the scope in which it was declared goes away, it has access to the original scope in which it was declared through its closure.

Let's continue to expand this example to understand how far you can go with closures:

```
var outer='outer';
var copy;
function outerFn() {
  var inner='inner';
  function innerFn(param){
    console.log(outer);
    console.log(inner);
    console.log(param);
    console.log(magic);
  }
  copy=innerFn;
}
console.log(magic); //ERROR: magic not defined
var magic="Magic";
outerFn();
copy("copy");
```

In the preceding example, we have added a few more things. First, we added a parameter to `innerFn()`—just to illustrate that parameters are also part of the closure. There are two important points that we want to highlight.

All variables in an outer scope are included even if they are declared after the function is declared. This makes it possible for the line, `console.log(magic)`, in `innerFn()`, to work.

However, the same line, `console.log(magic)`, in the global scope will fail because even within the same scope, variables not yet defined cannot be referenced.

All these examples were intended to convey a few concepts that govern how closures work. Closures are a prominent feature in the JavaScript language and you can see them in most libraries.

Let's look at some popular patterns around closures.

Timers and callbacks

In implementing timers or callbacks, you need to call the handler asynchronously, mostly at a later point in time. Due to the asynchronous calls, we need to access variables from outside the scope in such functions. Consider the following example:

```
function delay(message) {
  setTimeout( function timerFn(){
    console.log( message );
  }, 1000 );
}
delay( "Hello World" );
```

We pass the inner `timerFn()` function to the built-in library function, `setTimeout()`. However, `timerFn()` has a scope closure over the scope of `delay()`, and hence it can reference the variable message.

Private variables

Closures are frequently used to encapsulate some information as private variables. JavaScript does not allow such encapsulation found in programming languages such as Java or C++, but by using closures, we can achieve similar encapsulation:

```
function privateTest(){
  var points=0;
  this.getPoints=function(){
    return points;
  };
  this.score=function(){
    points++;
  };
}

var private = new privateTest();
private.score();
console.log(private.points); // undefined
console.log(private.getPoints());
```

In the preceding example, we are creating a function that we intend to call as a constructor. In this `privateTest()` function, we are creating a `var points=0` variable as a function-scoped variable. This variable is available only in `privateTest()`. Additionally, we create an accessor function (also called a getter)—`getPoints()`—this method allows us to read the value of only the points variable from outside `privateTest()`, making this variable private to the function. However, another method, `score()`, allows us to modify the value of the private point variable without directly accessing it from outside. This makes it possible for us to write code where a private variable is updated in a controlled fashion. This pattern can be very useful when you are writing libraries where you want to control how variables are accessed based on a contract and pre-established interface.

Loops and closures

Consider the following example of using functions inside loops:

```
for (var i=1; i<=5; i++) {
  setTimeout( function delay(){
    console.log( i );
  }, i*100);
}
```

This snippet should print `1`, `2`, `3`, `4`, and `5` on the console at an interval of 100 ms, right? Instead, it prints `6`, `6`, `6`, `6`, and `6` at an interval of 100 ms. Why is this happening? Here, we encounter a common issue with closures and looping. The `i` variable is being updated after the function is bound. This means that every bound function handler will always print the last value stored in `i`. In fact, the timeout function callbacks are running after the completion of the loop. This is such a common problem that JSLint will warn you if you try to use functions this way inside a loop.

How can we fix this behavior? We can introduce a function scope and local copy of the `i` variable in that scope. The following snippet shows you how we can do this:

```
for (var i=1; i<=5; i++) {
  (function(j){
    setTimeout( function delay(){
      console.log( j );
    }, j*100);
  })( i );
}
```

We pass the `i` variable and copy it to the `j` variable local to the IIFE. The introduction of an IIFE inside each iteration creates a new scope for each iteration and hence updates the local copy with the correct value.

Modules

Modules are used to mimic classes and focus on public and private access to variables and functions. Modules help in reducing the global scope pollution. Effective use of modules can reduce name collisions across a large code base. A typical format that this pattern takes is as follows:

```
Var moduleName=function() {
  //private state
  //private functions
  return {
     //public state
     //public variables
  }
}
```

There are two requirements to implement this pattern in the preceding format:

- There must be an outer enclosing function that needs to be executed at least once.
- This enclosing function must return at least one inner function. This is necessary to create a closure over the private state—without this, you can't access the private state at all.

Check the following example of a module:

```
var superModule = (function (){
  var secret = 'supersecretkey';
  var passcode = 'nuke';

  function getSecret() {
    console.log( secret );
  }

  function getPassCode() {
    console.log( passcode );
  }
```

```
    return {
      getSecret: getSecret,
      getPassCode: getPassCode
    };
  })();
  superModule.getSecret();
  superModule.getPassCode();
```

This example satisfies both the conditions. Firstly, we create an IIFE or a named function to act as an outer enclosure. The variables defined will remain private because they are scoped in the function. We return the public functions to make sure that we have a closure over the private scope. Using IIFE in the module pattern will actually result in a singleton instance of this function. If you want to create multiple instances, you can create named function expressions as part of the module as well.

We will keep exploring various facets of functional aspects of JavaScript and closures in particular. There can be a lot of imaginative uses of such elegant constructs. An effective way to understand various patterns is to study the code of popular libraries and practice writing these patterns in your code.

Stylistic considerations

As in the previous chapter, we will conclude this discussion with certain stylistic considerations. Again, these are generally accepted guidelines and not rules—feel free to deviate from them if you have reason to believe otherwise:

- Use function declarations instead of function expressions:

  ```
  // bad
  const foo = function () {
  };

  // good
  function foo() {
  }
  ```

- Never declare a function in a non-function block (if, while, and so on). Assign the function to a variable instead. Browsers allow you to do it, but they all interpret it differently.
- Never name a parameter `arguments`. This will take precedence over the `arguments` object that is given to every function scope.

Summary

In this chapter, we studied JavaScript functions. In JavaScript, functions play a critical role. We discussed how functions are created and used. We also discussed important ideas of closures and the scope of variables in terms of functions. We discussed functions as a way to create visibility classes and encapsulation.

In the next chapter, we will look at various data structures and data manipulation techniques in JavaScript.

3 Data Structures and Manipulation

Most of the time that you spend in programming, you do something to manipulate data. You process properties of data, derive conclusions based on the data, and change the nature of the data. In this chapter, we will take an exhaustive look at various data structures and data manipulation techniques in JavaScript. With the correct usage of these expressive constructs, your programs will be correct, concise, easy to read, and most probably faster. This will be explained with the help of the following topics:

- Regular expressions
- Exact match
- Match from a class of characters
- Repeated occurrences
- Beginning and end
- Backreferences
- Greedy and lazy quantifiers
- Arrays
- Maps
- Sets
- A matter of style

Regular expressions

If you are not familiar with regular expressions, I request you to spend time learning them. Learning and using regular expressions effectively is one of the most rewarding skills that you will gain. During most of the code review sessions, the first thing that I comment on is how a piece of code can be converted to a single line of **regular expression** (or **RegEx**). If you study popular JavaScript libraries, you will be surprised to see how ubiquitous RegEx are. Most seasoned engineers rely on RegEx primarily because once you know how to use them, they are concise and easy to test. However, learning RegEx will take a significant amount of effort and time. A regular expression is a way to express a pattern to match strings of text. The expression itself consists of terms and operators that allow us to define these patterns. We'll see what these terms and operators consist of shortly.

In JavaScript, there are two ways to create a regular expression: via a regular expression literal and constructing an instance of a `RegExp` object.

For example, if we wanted to create a RegEx that matches the string test exactly, we could use the following RegEx literal:

```
var pattern = /test/;
```

RegEx literals are delimited using forward slashes. Alternatively, we could construct a `RegExp` instance, passing the RegEx as a string:

```
var pattern = new RegExp("test");
```

Both of these formats result in the same RegEx being created in the variable pattern. In addition to the expression itself, there are three flags that can be associated with a RegEx:

- `i`: This makes the RegEx case-insensitive, so `/test/i` matches not only `test`, but also `Test`, `TEST`, `tEsT`, and so on.
- `g`: This matches all the instances of the pattern as opposed to the default of local, which matches the first occurrence only. More on this later.
- `m`: This allows matches across multiple lines that might be obtained from the value of a `textarea` element.

These flags are appended to the end of the literal (for example, `/test/ig`) or passed in a string as the second parameter to the `RegExp` constructor (`new RegExp("test", "ig")`).

The following example illustrates the various flags and how they affect the pattern match:

```
var pattern = /orange/;
console.log(pattern.test("orange")); // true

var patternIgnoreCase = /orange/i;
console.log(patternIgnoreCase.test("Orange")); // true

var patternGlobal = /orange/ig;
console.log(patternGlobal.test("Orange Juice")); // true
```

It isn't very exciting if we can just test whether the pattern matches a string. Let's see how we can express more complex patterns.

Exact match

Any sequence of characters that's not a special RegEx character or operator represents a character literal:

```
var pattern = /orange/;
```

We mean `o` followed by `r` followed by `a` followed by `n` followed by …—you get the point. We rarely use exact match when using RegEx because that is the same as comparing two strings. Exact match patterns are sometimes called simple patterns.

Match from a class of characters

If you want to match against a set of characters, you can place the set inside `[]`. For example, `[abc]` would mean any character `a`, `b`, or `c`:

```
var pattern = /[abc]/;
console.log(pattern.test('a')); //true
console.log(pattern.test('d')); //false
```

You can specify that you want to match anything but the pattern by adding a ^ (caret sign) at the beginning of the pattern:

```
var pattern = /[^abc]/;
console.log(pattern.test('a')); //false
console.log(pattern.test('d')); //true
```

One critical variation of this pattern is a range of values. If we want to match against a sequential range of characters or numbers, we can use the following pattern:

```
var pattern = /[0-5]/;
console.log(pattern.test(3)); //true
console.log(pattern.test(12345)); //true
console.log(pattern.test(9)); //false
console.log(pattern.test(6789)); //false
console.log(/[0123456789]/.test("This is year 2015")); //true
```

Special characters such as `$` and period (`.`) characters either represent matches to something other than themselves or operators that qualify the preceding term. In fact, we've already seen how `[`, `]`, `-`, and `^` characters are used to represent something other than their literal values.

How do we specify that we want to match a literal `[` or `$` or `^` or some other special character? Within a RegEx, the backslash character escapes whatever character follows it, making it a literal match term. So `\[` specifies a literal match to the `[` character rather than the opening of a character class expression. A double backslash (\\) matches a single backslash.

In the preceding examples, we saw the `test()` method that returns **true** or **false** based on the pattern matched. There are times when you want to access occurrences of a particular pattern. The `exec()` method comes in handy in such situations.

The `exec()` method takes a string as an argument and returns an array containing all matches. Consider the following example:

```
var strToMatch = 'A Toyota! Race fast, safe car! A Toyota!';
var regExAt = /Toy/;
var arrMatches = regExAt.exec(strToMatch);
console.log(arrMatches);
```

The output of this snippet would be **['Toy'];** if you want all the instances of the pattern `Toy`, you can use the `g` (global) flag as follows:

```
var strToMatch = 'A Toyota! Race fast, safe car! A Toyota!';
var regExAt = /Toy/g;
var arrMatches = regExAt.exec(strToMatch);
console.log(arrMatches);
```

This will return all the occurrences of the word `oyo` from the original text. The String object contains the `match()` method that has similar functionality of the `exec()` method. The `match()` method is called on a String object and the RegEx is passed to it as a parameter. Consider the following example:

```
var strToMatch = 'A Toyota! Race fast, safe car! A Toyota!';
var regExAt = /Toy/;
var arrMatches = strToMatch.match(regExAt);
console.log(arrMatches);
```

In this example, we are calling the `match()` method on the String object. We pass the RegEx as a parameter to the `match()` method. The results are the same in both these cases.

The other String object method is `replace()`. It replaces all the occurrences of a substring with a different string:

```
var strToMatch = 'Blue is your favorite color ?';
var regExAt = /Blue/;
console.log(strToMatch.replace(regExAt, "Red"));
//Output- "Red is your favorite color ?"
```

It is possible to pass a function as a second parameter of the `replace()` method. The `replace()` function takes the matching text as a parameter and returns the text that is used as a replacement:

```
var strToMatch = 'Blue is your favorite color ?';
var regExAt = /Blue/;
console.log(strToMatch.replace(regExAt, function(matchingText){
  return 'Red';
}));
//Output- "Red is your favorite color ?"
```

The String object's `split()` method also takes a RegEx parameter and returns an array containing all the substrings generated after splitting the original string:

```
var sColor = 'sun,moon,stars';
var reComma = /\,/;
console.log(sColor.split(reComma));
//Output - ["sun", "moon", "stars"]
```

We need to add a backslash before the comma because a comma is treated specially in RegEx and we need to escape it if we want to use it literally.

Using simple character classes, you can match multiple patterns. For example, if you want to match `cat`, `bat`, and `fat`, the following snippet shows you how to use simple character classes:

```
var strToMatch = 'wooden bat, smelly Cat,a fat cat';
var re = /[bcf]at/gi;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
//["bat", "Cat", "fat", "cat"]
```

As you can see, this variation opens up possibilities to write concise RegEx patterns. Take the following example:

```
var strToMatch = 'i1,i2,i3,i4,i5,i6,i7,i8,i9';
var re = /i[0-5]/gi;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
//["i1", "i2", "i3", "i4", "i5"]
```

In this example, we are matching the numeric part of the matching string with a range `[0-5]`, hence we get a match from `i0` to `i5`. You can also use the negation class ^ to filter the rest of the matches:

```
var strToMatch = 'i1,i2,i3,i4,i5,i6,i7,i8,i9';
var re = /i[^0-5]/gi;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
//["i6", "i7", "i8", "i9"]
```

Observe how we are negating only the range clause and not the entire expression.

Several character groups have shortcut notations. For example, the shortcut `\d` means the same thing as `[0-9]`:

Notation	**Meaning**
`\d`	Any digit character
`\w`	An alphanumeric character (word character)
`\s`	Any whitespace character (space, tab, newline, and similar)
`\D`	A character that is not a digit
`\W`	A non-alphanumeric character
`\S`	A non-whitespace character
`.`	Any character except for newline

These shortcuts are valuable in writing concise RegEx. Consider this example:

```
var strToMatch = '123-456-7890';
var re = /[0-9][0-9][0-9]-[0-9][0-9][0-9]/;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
//["123-456"]
```

This expression definitely looks a bit strange. We can replace `[0-9]` with `\d` and make this a bit more readable:

```
var strToMatch = '123-456-7890';
var re = /\d\d\d-\d\d\d/;
var arrMatches = strToMatch.match(re);
console.log(arrMatches);
//["123-456"]
```

However, you will soon see that there are even better ways to do something like this.

Repeated occurrences

So far, we saw how we can match fixed characters or numeric patterns. Most often, you want to handle certain repetitive natures of patterns also. For example, if I want to match 4 `as`, I can write `/aaaa/`, but what if I want to specify a pattern that can match any number of `as`?

Regular expressions provide you with a wide variety of repetition quantifiers. Repetition quantifiers let us specify how many times a particular pattern can occur. We can specify fixed values (characters should appear *n* times) and variable values (characters can appear at least *n* times till they appear *m* times). The following table lists the various repetition quantifiers:

- `?`: Either 0 or 1 occurrence (marks the occurrence as optional)
- `*`: 0 or more occurrences
- `+`: 1 or more occurrences
- `{n}`: Exactly `n` occurrences
- `{n,m}`: Occurrences between `n` and `m`
- `{n,}`: At least an `n` occurrence
- `{,n}`: 0 to `n` occurrences

In the following example, we create a pattern where the character `u` is optional (has 0 or 1 occurrence):

```
var str = /behaviou?r/;
console.log(str.test("behaviour"));
// true
console.log(str.test("behavior"));
// true
```

It helps to read the `/behaviou?r/` expression as 0 or 1 occurrences of character `u`. The repetition quantifier succeeds the character that we want to repeat. Let's try out some more examples:

```
console.log(/'\d+'/.test("'123'")); // true
```

You should read and interpret the `\d+` expression as `'` is a literal character match, `\d` matches characters `[0-9]`, the `+` quantifier will allow one or more occurrences, and `'` is a literal character match.

You can also group character expressions using `()`. Observe the following example:

```
var heartyLaugh = /Ha+(Ha+)+/i;
console.log(heartyLaugh.test("HaHaHaHaHaHaHaaaaaaaaaaa"));
//true
```

Let's break the preceding expression into smaller chunks to understand what is going on in here:

- `H`: literal character match
- `a+`: 1 or more occurrences of character `a`
- `(`: start of the expression group
- `H`: literal character match
- `a+`: 1 or more occurrences of character `a`
- `)`: end of expression group
- `+`: 1 or more occurrences of expression group (`Ha+`)

Now it is easier to see how the grouping is done. If we have to interpret the expression, it is sometimes helpful to read out the expression, as shown in the preceding example.

Often, you want to match a sequence of letters or numbers on their own and not just as a substring. This is a fairly common use case when you are matching words that are not just part of any other words. We can specify the word boundaries by using the `\b` pattern. The word boundary with `\b` matches the position where one side is a word character (letter, digit, or underscore) and the other side is not. Consider the following examples.

The following is a simple literal match. This match will also be successful if `cat` is part of a substring:

```
console.log(/cat/.test('a black cat')); //true
```

However, in the following example, we define a word boundary by indicating `\b` before the word `cat`—this means that we want to match only if `cat` is a word and not a substring. The boundary is established before `cat`, and hence a match is found on the text, `a black cat`:

```
console.log(/\bcat/.test('a black cat')); //true
```

When we use the same boundary with the word `tomcat`, we get a failed match because there is no word boundary before `cat` in the word `tomcat`:

```
console.log(/\bcat/.test('tomcat')); //false
```

There is a word boundary after the string `cat` in the word `tomcat`, hence the following is a successful match:

```
console.log(/cat\b/.test('tomcat')); //true
```

In the following example, we define the word boundary before and after the word `cat` to indicate that we want `cat` to be a standalone word with boundaries before and after:

```
console.log(/\bcat\b/.test('a black cat')); //true
```

Based on the same logic, the following match fails because there are no boundaries before and after `cat` in the word `concatenate`:

```
console.log(/\bcat\b/.test("concatenate")); //false
```

The `exec()` method is useful in getting information about the match found because it returns an object with information about the match. The object returned from `exec()` has an `index` property that tells us where the successful match begins in the string. This is useful in many ways:

```
var match = /\d+/.exec("There are 100 ways to do this");
console.log(match);
// ["100"]
console.log(match.index);
// 10
```

Alternatives – OR

Alternatives can be expressed using the | (pipe) character. For example, `/a|b/` matches either the `a` or `b` character, and `/(ab)+|(cd)+/` matches one or more occurrences of either `ab` or `cd`.

Beginning and end

Frequently, we may wish to ensure that a pattern matches at the beginning of a string or perhaps at the end of a string. The caret character, when used as the first character of the RegEx, anchors the match at the beginning of the string such that `/^test/` matches only if the test substring appears at the beginning of the string being matched. Similarly, the dollar sign (`$`) signifies that the pattern must appear at the end of the string: `/test$/`.

Using both `^` and `$` indicates that the specified pattern must encompass the entire candidate string: `/^test$/`.

Backreferences

After an expression is evaluated, each group is stored for later use. These values are known as backreferences. Backreferences are created and numbered by the order in which opening parenthesis characters are encountered going from left to right. You can think of backreferences as the portions of a string that are successfully matched against terms in the regular expression.

The notation for a backreference is a backslash followed by the number of the capture to be referenced, beginning with 1, such as `\1`, `\2`, and so on.

An example could be `/^([XYZ])a\1/`, which matches a string that starts with any of the `X`, `Y`, or `Z` characters followed by an `a` and followed by whatever character matched the first capture. This is very different from `/[XYZ] a[XYZ]/`. The character following `a` can't be any of `X`, or `Y`, or `Z`, but must be whichever one of those that triggered the match for the first character. Backreferences are used with String's `replace()` method using the special character sequences, `$1`, `$2`, and so on. Suppose that you want to change the `1234 5678` string to `5678 1234`. The following code accomplishes this:

```
var orig = "1234 5678";
var re = /(\d{4}) (\d{4})/;
var modifiedStr = orig.replace(re, "$2 $1");
console.log(modifiedStr); //outputs "5678 1234"
```

In this example, the regular expression has two groups each with four digits. In the second argument of the `replace()` method, `$2` is equal to `5678` and `$1` is equal to `1234`, corresponding to the order in which they appear in the expression.

Greedy and lazy quantifiers

All the quantifiers that we discussed so far are greedy. A greedy quantifier starts looking at the entire string for a match. If there are no matches, it removes the last character in the string and reattempts the match. If a match is not found again, the last character is again removed and the process is repeated until a match is found or the string is left with no characters.

The `\d+` pattern, for example, will match one or more digits. For example, if your string is `123`, a greedy match would match `1`, `12`, and `123`. Greedy pattern `h.+l` would match `hell` in a string `hello`—which is the longest possible string match. As `\d+` is greedy, it will match as many digits as possible and hence the match would be `123`.

In contrast to greedy quantifiers, a lazy quantifier matches as few of the quantified tokens as possible. You can add a question mark (`?`) to the regular expression to make it lazy. A lazy pattern `h.?l` would match `hel` in the string `hello`—which is the shortest possible string.

The `\w*?X` pattern will match zero or more words and then match an `X`. However, a question mark after `*` indicates that as few characters as possible should be matched. For an `abcXXX` string, the match can be `abcX`, `abcXX`, or `abcXXX`. Which one should be matched? As `*?` is lazy, as few characters as possible are matched and hence the match is `abcX`.

With this necessary information, let's try to solve some common problems using regular expressions.

Removing extra white space from the beginning and end of a string is a very common use case. As a String object did not have the `trim()` method until recently, several JavaScript libraries provide and use an implementation of string trimming for older browsers that don't have the `String.trim()` method. The most commonly used approach looks something like the following code:

```
function trim(str) {
  return (str || "").replace(/^\s+|\s+$/g, "");
}
console.log("--"+trim("   test    ")+"--");
//"--test--"
```

What if we want to replace repeated whitespaces with a single whitespace?

```
re=/\s+/g;
console.log('There are    a lot      of spaces'.replace(re,' '));
//"There are a lot of spaces"
```

In the preceding snippet, we are trying to match one or more space character sequences and replacing them with a single space.

As you can see, regular expressions can prove to be a Swiss army knife in your JavaScript arsenal. Careful study and practice will be extremely rewarding for you in the long run.

Arrays

An array is an ordered set of values. You can refer to the array elements with a name and index. These are the three ways to create arrays in JavaScript:

```
var arr = new Array(1,2,3);
var arr = Array(1,2,3);
var arr = [1,2,3];
```

When these values are specified, the array is initialized with them as the array's elements. An array's `length` property is equal to the number of arguments. The bracket syntax is called an array literal. It's a shorter and preferred way to initialize arrays.

You have to use the array literal syntax if you want to initialize an array with a single element and the element happens to be a number. If you pass a single number value to the `Array()` constructor or function, JavaScript considers this parameter as the length of the array, not as a single element:

```
var arr = [10];
var arr = Array(10); // Creates an array with no element, but with
  arr.length set to 10
// The above code is equivalent to
var arr = [];
arr.length = 10;
```

JavaScript does not have an explicit array data type. However, you can use the predefined `Array` object and its methods to work with arrays in your applications. The `Array` object has methods to manipulate arrays in various ways, such as joining, reversing, and sorting them. It has a property to determine the array length and other properties for use with regular expressions.

You can populate an array by assigning values to its elements:

```
var days = [];
days[0] = "Sunday";
days[1] = "Monday";
```

You can also populate an array when you create it:

```
var arr_generic = new Array("A String", myCustomValue, 3.14);
var fruits = ["Mango", "Apple", "Orange"]
```

In most languages, the elements of an array are all required to be of the same type. JavaScript allows an array to contain any type of values:

```
var arr = [
  'string', 42.0, true, false, null, undefined,
  ['sub', 'array'], {object: true}, NaN
];
```

You can refer to elements of an `Array` using the element's index number. For example, suppose you define the following array:

```
var days = ["Sunday", "Monday", "Tuesday"]
```

You then refer to the first element of the array as `colors[0]` and the second element of the array as `colors[1]`. The index of the elements starts with `0`.

JavaScript internally stores array elements as standard object properties, using the array index as the property name. The `length` property is different. The `length` property always returns the index of the last element plus one. As we discussed, JavaScript array indexes are 0-based: they start at `0`, not `1`. This means that the `length` property will be one more than the highest index stored in the array:

```
var colors = [];
colors[30] = ['Green'];
console.log(colors.length); // 31
```

You can also assign to the `length` property. Writing a value that is shorter than the number of stored items truncates the array; writing `0` empties it entirely:

```
var colors = ['Red', 'Blue', 'Yellow'];
console.log(colors.length); // 3
colors.length = 2;
console.log(colors); // ["Red","Blue"] - Yellow has been removed
colors.length = 0;
console.log(colors); // [] the colors array is empty
colors.length = 3;
console.log(colors); // [undefined, undefined, undefined]
```

If you query a non-existent array index, you get `undefined`.

A common operation is to iterate over the values of an array, processing each one in some way. The simplest way to do this is as follows:

```
var colors = ['red', 'green', 'blue'];
for (var i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}
```

The `forEach()` method provides another way of iterating over an array:

```
var colors = ['red', 'green', 'blue'];
colors.forEach(function(color) {
  console.log(color);
});
```

The function passed to `forEach()` is executed once for every item in the array, with the array item passed as the argument to the function. Unassigned values are not iterated in a `forEach()` loop.

The `Array` object has a bunch of useful methods. These methods allow the manipulation of the data stored in the array.

The `concat()` method joins two arrays and returns a new array:

```
var myArray = new Array("33", "44", "55");
myArray = myArray.concat("3", "2", "1");
console.log(myArray);
// ["33", "44", "55", "3", "2", "1"]
```

The `join()` method joins all the elements of an array into a string. This can be useful while processing a list. The default delimiter is a comma (`,`):

```
var myArray = new Array('Red','Blue','Yellow');
var list = myArray.join(" ~ ");
console.log(list);
//"Red ~ Blue ~ Yellow"
```

The `pop()` method removes the last element from an array and returns that element. This is analogous to the `pop()` method of a stack:

```
var myArray = new Array("1", "2", "3");
var last = myArray.pop();
// myArray = ["1", "2"], last = "3"
```

The `push()` method adds one or more elements to the end of an array and returns the resulting length of the array:

```
var myArray = new Array("1", "2");
myArray.push("3");
// myArray = ["1", "2", "3"]
```

The `shift()` method removes the first element from an array and returns that element:

```
var myArray = new Array ("1", "2", "3");
var first = myArray.shift();
// myArray = ["2", "3"], first = "1"
```

The `unshift()` method adds one or more elements to the front of an array and returns the new length of the array:

```
var myArray = new Array ("1", "2", "3");
myArray.unshift("4", "5");
// myArray = ["4", "5", "1", "2", "3"]
```

The `reverse()` method reverses or transposes the elements of an array—the first array element becomes the last and the last becomes the first:

```
var myArray = new Array ("1", "2", "3");
myArray.reverse();
// transposes the array so that myArray = [ "3", "2", "1" ]
```

The `sort()` method sorts the elements of an array:

```
var myArray = new Array("A", "C", "B");
myArray.sort();
// sorts the array so that myArray = [ "A","B","c" ]
```

The `sort()` method can optionally take a callback function to define how the elements are compared. The function compares two values and returns one of three values. Let us study the following functions:

- `indexOf(searchElement[, fromIndex])`: This searches the array for `searchElement` and returns the index of the first match:

  ```
  var a = ['a', 'b', 'a', 'b', 'a','c','a'];
  console.log(a.indexOf('b')); // 1
  // Now try again, starting from after the last match
  console.log(a.indexOf('b', 2)); // 3
  console.log(a.indexOf('1')); // -1, 'q' is not found
  ```

- `lastIndexOf(searchElement[, fromIndex])`: This works like `indexOf()`, but only searches backwards:

  ```
  var a = ['a', 'b', 'c', 'd', 'a', 'b'];
  console.log(a.lastIndexOf('b')); //  5
  // Now try again, starting from before the last match
  console.log(a.lastIndexOf('b', 4)); //  1
  console.log(a.lastIndexOf('z')); //  -1
  ```

Now that we have covered JavaScript arrays in depth, let me introduce you to a fantastic library called **Underscore.js** (`http://underscorejs.org/`). Underscore.js provides a bunch of exceptionally useful functional programming helpers to make your code even more clear and functional.

We will assume that you are familiar with **Node.js**; in this case, install Underscore.js via npm:

```
npm install underscore
```

As we are installing Underscore as a Node module, we will test all the examples by typing them in a `.js` file and running the file on Node.js. You can install Underscore using **Bower** also.

Like jQuery's `$` module, Underscore comes with a _ module defined. You will call all functions using this module reference.

Type the following code in a text file and name it `test_.js`:

```
var _ = require('underscore');
function print(n){
  console.log(n);
}
_.each([1, 2, 3], print);
//prints 1 2 3
```

This can be written as follows, without using `each()` function from underscore library:

```
var myArray = [1,2,3];
var arrayLength = myArray.length;
for (var i = 0; i < arrayLength; i++) {
  console.log(myArray[i]);
}
```

What you see here is a powerful functional construct that makes the code much more elegant and concise. You can clearly see that the traditional approach is verbose. Many languages such as Java suffer from this verbosity. They are slowly embracing functional paradigms. As JavaScript programmers, it is important for us to incorporate these ideas into our code as much as possible.

The `each()` function we saw in the preceding example iterates over a list of elements, yielding each to an iteratee function in turn. Each invocation of iteratee is called with three arguments (element, index, and list). In the preceding example, the `each()` function iterates over the array `[1,2,3]`, and for each element in the array, the `print` function is called with the array element as the parameter. This is a convenient alternative to the traditional looping mechanism to access all the elements in an array.

The `range()` function creates lists of integers. The start value, if omitted, defaults to `0` and step defaults to `1`. If you'd like a negative range, use a negative step:

```
var _ = require('underscore');
console.log(_.range(10));
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
console.log(_.range(1, 11));
//[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
console.log(_.range(0, 30, 5));
//[ 0, 5, 10, 15, 20, 25 ]
console.log(_.range(0, -10, -1));
//[ 0, -1, -2, -3, -4, -5, -6, -7, -8, -9 ]
console.log(_.range(0));
//[]
```

By default, `range()` populates the array with integers, but with a little trick, you can populate other data types also:

```
console.log(_.range(3).map(function () { return 'a' }) );
[ 'a', 'a', 'a' ]
```

This is a fast and convenient way to create and initialize an array with values. We frequently do this by traditional loops.

The `map()` function produces a new array of values by mapping each value in the list through a transformation function. Consider the following example:

```
var _ = require('underscore');
console.log(_.map([1, 2, 3], function(num){ return num * 3; }));
//[3,6,9]
```

The `reduce()` function reduces a list of values to a single value. The initial state is passed by the iteratee function and each successive step is returned by the iteratee. The following example shows the usage:

```
var _ = require('underscore');
var sum = _.reduce([1, 2, 3], function(memo, num){
  console.log(memo,num);return memo + num; }, 0);
console.log(sum);
```

In this example, the line, `console.log(memo,num);`, is just to make the idea clear. The output will be as follows:

```
0 1
1 2
3 3
6
```

The final output is a sum of *1+2+3=6*. As you can see, two values are passed to the iteratee function. On the first iteration, we call the iteratee function with two values `(0,1)`—the value of the `memo` is defaulted in the call to the `reduce()` function and `1` is the first element of the list. In the function, we sum `memo` and `num` and return the intermediate `sum`, which will be used by the `iterate()` function as a `memo` parameter—eventually, the `memo` will have the accumulated `sum`. This concept is important to understand how the intermediate states are used to calculate eventual results.

The `filter()` function iterates through the entire list and returns an array of all the elements that pass the condition. Take a look at the following example:

```
var _ = require('underscore');
var evens = _.filter([1, 2, 3, 4, 5, 6], function(num){
  return num % 2 == 0; });
console.log(evens);
```

The `filter()` function's iteratee function should return a truth value. The resulting `evens` array contains all the elements that satisfy the truth test.

The opposite of the `filter()` function is `reject()`. As the name suggests, it iterates through the list and ignores elements that satisfy the truth test:

```
var _ = require('underscore');
var odds = _.reject([1, 2, 3, 4, 5, 6], function(num){
  return num % 2 == 0; });
console.log(odds);
//[ 1, 3, 5 ]
```

We are using the same code as the previous example but using the `reject()` method instead of `filter()`—the result is exactly the opposite.

The `contains()` function is a useful little function that returns `true` if the value is present in the list; otherwise, returns `false`:

```
var _ = require('underscore');
console.log(_.contains([1, 2, 3], 3));
//true
```

One very useful function that I have grown fond of is `invoke()`. It calls a specific function on each element in the list. I can't tell you how many times I have used it since I stumbled upon it. Let us study the following example:

```
var _ = require('underscore');
console.log(_.invoke([[5, 1, 7], [3, 2, 1]], 'sort'));
//[ [ 1, 5, 7 ], [ 1, 2, 3 ] ]
```

In this example, the `sort()` method of the `Array` object is called for each element in the array. Note that this would fail:

```
var _ = require('underscore');
console.log(_.invoke(["new","old","cat"], 'sort'));
//[ undefined, undefined, undefined ]
```

This is because the `sort` method is not part of the String object. This, however, would work perfectly:

```
var _ = require('underscore');
console.log(_.invoke(["new","old","cat"], 'toUpperCase'));
//[ 'NEW', 'OLD', 'CAT' ]
```

This is because `toUpperCase()` is a String object method and all elements of the list are of the String type.

The `uniq()` function returns the array after removing all duplicates from the original one:

```
var _ = require('underscore');
var uniqArray = _.uniq([1,1,2,2,3]);
console.log(uniqArray);
//[1,2,3]
```

The `partition()` function splits the array into two; one whose elements satisfy the predicate and the other whose elements don't satisfy the predicate:

```
var _ = require('underscore');
function isOdd(n){
  return n%2==0;
}
console.log(_.partition([0, 1, 2, 3, 4, 5], isOdd));
//[ [ 0, 2, 4 ], [ 1, 3, 5 ] ]
```

The `compact()` function returns a copy of the array without all falsy values (false, null, 0, "", undefined, and NaN):

```
console.log(_.compact([0, 1, false, 2, '', 3]));
```

This snippet will remove all falsy values and return a new array with elements `[1,2,3]`—this is a helpful method to eliminate any value from a list that can cause runtime exceptions.

The `without()` function returns a copy of the array with all instances of the specific values removed:

```
var _ = require('underscore');
console.log(_.without([1,2,3,4,5,6,7,8,9,0,1,2,0,0,1,1],0,1,2));
//[ 3, 4, 5, 6, 7, 8, 9 ]
```

Maps

ECMAScript 6 introduces maps. A map is a simple key-value map and can iterate its elements in the order of their insertion. The following snippet shows some methods of the `Map` type and their usage:

```
var founders = new Map();
founders.set("facebook", "mark");
founders.set("google", "larry");
founders.size; // 2
founders.get("twitter"); // undefined
founders.has("yahoo"); // false

for (var [key, value] of founders) {
  console.log(key + " founded by " + value);
}
// "facebook founded by mark"
// "google founded by larry"
```

Sets

ECMAScript 6 introduces sets. Sets are collections of values and can be iterated in the order of the insertion of their elements. An important characteristic about sets is that a value can occur only once in a set.

The following snippet shows some basic operations on sets:

```
var mySet = new Set();
mySet.add(1);
mySet.add("Howdy");
mySet.add("foo");

mySet.has(1); // true
mySet.delete("foo");
mySet.size; // 2

for (let item of mySet) console.log(item);
// 1
// "Howdy"
```

We discussed briefly that JavaScript arrays are not really arrays in a traditional sense. In JavaScript, arrays are objects that have the following characteristics:

- The `length` property
- The functions that inherit from `Array.prototype` (we will discuss this in the next chapter)
- Special handling for keys that are numeric keys

When we write an array index as numbers, they get converted to strings—`arr[0]` internally becomes `arr["0"]`. Due to this, there are a few things that we need to be aware of when we use JavaScript arrays:

- Accessing array elements by an index is not a constant time operation as it is in, say, C. As arrays are actually key-value maps, the access will depend on the layout of the map and other factors (collisions and others).
- JavaScript arrays are sparse (most of the elements have the default value), which means that the array can have gaps in it. To understand this, look at the following snippet:

  ```
  var testArr=new Array(3);
  console.log(testArr);
  ```

  You will see the output as `[undefined, undefined, undefined]`—`undefined` is the default value stored on the array element.

Consider the following example:

```
var testArr=[];
testArr[3] = 10;
testArr[10] = 3;
console.log(testArr);
// [undefined, undefined, undefined, 10, undefined, undefined,
  undefined, undefined, undefined, undefined, 3]
```

You can see that there are gaps in this array. Only two elements have elements and the rest are gaps with the default value. Knowing this helps you in a couple of things. Using the `for...in` loop to iterate an array can result in unexpected results. Consider the following example:

```
var a = [];
a[5] = 5;
for (var i=0; i<a.length; i++) {
  console.log(a[i]);
}
```

```
// Iterates over numeric indexes from 0 to 5
// [undefined,undefined,undefined,undefined,undefined,5]

for (var x in a) {
  console.log(x);
}
// Shows only the explicitly set index of "5", and ignores 0-4
```

A matter of style

Like the previous chapters, we will spend some time discussing the style considerations while creating arrays.

- Use the literal syntax for array creation:

  ```
  // bad
  const items = new Array();
  // good
  const items = [];
  ```

- Use `Array#push` instead of a direct assignment to add items to an array:

  ```
  const stack = [];
  // bad
  stack[stack.length] = 'pushme';
  // good
  stack.push('pushme');
  ```

Summary

As JavaScript matures as a language, its tool chain also becomes more robust and effective. It is rare to see seasoned programmers staying away from libraries such as Underscore.js. As we see more advanced topics, we will continue to explore more such versatile libraries that can make your code compact, more readable, and performant. We looked at regular expressions—they are first-class objects in JavaScript. Once you start understanding `RegExp`, you will soon find yourself using more of them to make your code concise. In the next chapter, we will look at JavaScript Object notation and how JavaScript prototypal inheritance is a new way of looking at object-oriented programming.

4 Object-Oriented JavaScript

JavaScript's most fundamental data type is the Object data type. JavaScript objects can be seen as mutable key-value-based collections. In JavaScript, arrays, functions, and RegExp are objects while numbers, strings, and Booleans are object-like constructs that are immutable but have methods. In this chapter, you will learn the following topics:

- Understanding objects
- Instance properties versus prototype properties
- Inheritance
- Getters and setters

Understanding objects

Before we start looking at how JavaScript treats objects, we should spend some time on an object-oriented paradigm. Like most programming paradigms, **object-oriented programming** (**OOP**) also emerged from the need to manage complexity. The main idea is to divide the entire system into smaller pieces that are isolated from each other. If these small pieces can hide as many implementation details as possible, they become easy to use. A classic car analogy will help you understand this very important point about OOP.

When you drive a car, you operate on the interface—the steering, clutch, brake, and accelerator. Your view of using the car is limited by this interface, which makes it possible for us to drive the car. This interface is essentially hiding all the complex systems that really drive the car, such as the internal functioning of its engine, its electronic system, and so on. As a driver, you don't bother about these complexities. A similar idea is the primary driver of OOP. An object hides the complexities of how to implement a particular functionality and exposes a limited interface to the outside world. All other systems can use this interface without really bothering about the internal complexity that is hidden from view. Additionally, an object usually hides its internal state from other objects and prevents its direct modification. This is an important aspect of OOP.

In a large system where a lot of objects call other objects' interfaces, things can go really bad if you allow them to modify the internal state of such objects. OOP operates on the idea that the state of an object is inherently hidden from the outside world and it can be changed only via controlled interface operations.

OOP was an important idea and a definite step forward from the traditional structured programming. However, many feel that OOP is overdone. Most OOP systems define complex and unnecessary class and type hierarchies. Another big drawback was that in the pursuit of hiding the state, OOP considered the object state almost immaterial. Though hugely popular, OOP was clearly flawed in many areas. Still, OOP did have some very good ideas, especially hiding the complexity and exposing only the interface to the outside world. JavaScript picked up a few good ideas and built its object model around them. Luckily, this makes JavaScript objects very versatile. In their seminal work, *Design Patterns: Elements of Reusable Object-Oriented Software*, the *Gang of Four* gave two fundamental principles of a better object-oriented design:

- Program to an interface and not to an implementation
- Object composition over class inheritance

These two ideas are really against how classical OOP operates. The classical style of inheritance operates on inheritance that exposes parent classes to all child classes. Classical inheritance tightly couples children to its parents. There are mechanisms in classical inheritance to solve this problem to a certain extent. If you are using classical inheritance in a language such as Java, it is generally advisable to *program to an interface, not an implementation*. In Java, you can write a decoupled code using interfaces:

```
//programming to an interface 'List' and not implementation
  'ArrayList'
List theList = new ArrayList();
```

Instead of programming to an implementation, you can perform the following:

```
ArrayList theList = new ArrayList();
```

How does programming to an interface help? When you program to the `List` interface, you can call methods only available to the `List` interface and nothing specific to `ArrayList` can be called. Programming to an interface gives you the liberty to change your code and use any other specific child of the `List` interface. For example, I can change my implementation and use `LinkedList` instead of `ArrayList`. You can change your variable to use `LinkedList` instead:

```
List theList = new LinkedList();
```

The beauty of this approach is that if you are using the `List` at 100 places in your program, you don't have to worry about changing the implementation at all these places. As you were programming to the interface and not to the implementation, you were able to write a loosely coupled code. This is an important principle when you are using classical inheritance.

Classical inheritance also has a limitation where you can only enhance the child class within the limit of the parent classes. You can't fundamentally differ from what you have got from the ancestors. This inhibits reuse. Classical inheritance has several other problems as follows:

- Inheritance introduces tight coupling. Child classes have knowledge about their ancestors. This tightly couples a child class with its parent.
- When you subclass from a parent, you don't have a choice to select what you want to inherit and what you don't. *Joe Armstrong* (the inventor of **Erlang**) explains this situation very well—his now famous quote:

  "The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."

Behavior of JavaScript objects

With this background, let's explore how JavaScript objects behave. In broad terms, an object contains properties, defined as a key-value pair. A property key (name) can be a string and the value can be any valid JavaScript value. You can create objects using object literals. The following snippet shows you how object literals are created:

```
var nothing = {};
var author = {
  "firstname": "Douglas",
  "lastname": "Crockford"
}
```

A property's name can be any string or an empty string. You can omit quotes around the property name if the name is a legal JavaScript name. So quotes are required around `first-name` but are optional around `firstname`. Commas are used to separate the pairs. You can nest objects as follows:

```
var author = {
  firstname : "Douglas",
  lastname : "Crockford",
  book : {
    title:"JavaScript- The Good Parts",
    pages:"172"
  }
};
```

Properties of an object can be accessed by using two notations: the array-like notation and dot notation. According to the array-like notation, you can retrieve the value from an object by wrapping a string expression in `[]`. If the expression is a valid JavaScript name, you can use the dot notation using `.` instead. Using `.` is a preferred method of retrieving values from an object:

```
console.log(author['firstname']); //Douglas
console.log(author.lastname);     //Crockford
console.log(author.book.title);   // JavaScript- The Good Parts
```

You will get an `undefined` error if you attempt to retrieve a non-existent value. The following would return `undefined`:

```
console.log(author.age);
```

A useful trick is to use the `||` operator to fill in default values in this case:

```
console.log(author.age || "No Age Found");
```

You can update values of an object by assigning a new value to the property:

```
author.book.pages = 190;
console.log(author.book.pages); //190
```

If you observe closely, you will realize that the object literal syntax that you see is very similar to the JSON format.

Methods are properties of an object that can hold function values as follows:

```
var meetingRoom = {};
meetingRoom.book = function(roomId){
  console.log("booked meeting room -"+roomId);
}
meetingRoom.book("VL");
```

Prototypes

Apart from the properties that we add to an object, there is one default property for almost all objects, called a **prototype**. When an object does not have a requested property, JavaScript goes to its prototype to look for it. The `Object.getPrototypeOf()` function returns the prototype of an object.

Many programmers consider prototypes closely related to objects' inheritance—they are indeed a way of defining object types—but fundamentally, they are closely associated with functions.

Prototypes are used as a way to define properties and functions that will be applied to instances of objects. The prototype's properties eventually become properties of the instantiated objects. Prototypes can be seen as blueprints for object creation. They can be seen as analogous to classes in object-oriented languages. Prototypes in JavaScript are used to write a classical style object-oriented code and mimic classical inheritance. Let's revisit our earlier example:

```
var author = {};
author.firstname = 'Douglas';
author.lastname = 'Crockford';
```

In the preceding code snippet, we are creating an empty object and assigning individual properties. You will soon realize that this is not a very standard way of building objects. If you know OOP already, you will immediately see that there is no encapsulation and the usual class structure. JavaScript provides a way around this. You can use the `new` operator to instantiate an object via constructors. However, there is no concept of a class in JavaScript, and it is important to note that the `new` operator is applied to the constructor function. To clearly understand this, let's look at the following example:

```
//A function that returns nothing and creates nothing
function Player() {}
```

```
//Add a function to the prototype property of the function
Player.prototype.usesBat = function() {
  return true;
}

//We call player() as a function and prove that nothing happens
var crazyBob = Player();
if(crazyBob === undefined){
  console.log("CrazyBob is not defined");
}

//Now we call player() as a constructor along with 'new'
//1. The instance is created
//2. method usesBat() is derived from the prototype of the function
var swingJay = new Player();
if(swingJay && swingJay.usesBat && swingJay.usesBat()){
  console.log("SwingJay exists and can use bat");
}
```

In the preceding example, we have a `player()` function that does nothing. We invoke it in two different ways. The first call of the function is as a normal function and second call is as a constructor—note the use of the `new()` operator in this call. Once the function is defined, we add a `usesBat()` method to it. When this function is called as a normal function, the object is not instantiated and we see `undefined` assigned to `crazyBob`. However, when we call this function with the `new` operator, we get a fully instantiated object, `swingJay`.

Instance properties versus prototype properties

Instance properties are the properties that are part of the object instance itself, as shown in the following example:

```
function Player() {
  this.isAvailable = function() {
    return "Instance method says - he is hired";
  };
}
Player.prototype.isAvailable = function() {
  return "Prototype method says - he is Not hired";
};
var crazyBob = new Player();
console.log(crazyBob.isAvailable());
```

When you run this example, you will see that **Instance method says - he is hired** is printed. The `isAvailable()` function defined in the `Player()` function is called an instance of `Player`. This means that apart from attaching properties via the prototype, you can use the this keyword to initialize properties in a constructor. When we have the same functions defined as an instance property and also as a prototype, the instance property takes precedence. The rules governing the precedence of the initialization are as follows:

- Properties are tied to the object instance from the prototype
- Properties are tied to the object instance in the constructor function

This example brings us to the use of the `this` keyword. It is easy to get confused by the `this` keyword because it behaves differently in JavaScript. In other OO languages such as Java, the `this` keyword refers to the current instance of the class. In JavaScript, the value of `this` is determined by the invocation context of a function and where it is called. Let's see how this behavior needs to be carefully understood:

- When `this` is used in a global context: When `this` is called in a global context, it is bound to the global context. For example, in the case of a browser, the global context is usually `window`. This is true for functions also. If you use `this` in a function that is defined in the global context, it is still bound to the global context because the function is part of the global context:

  ```
  function globalAlias(){
    return this;
  }
  console.log(globalAlias()); //[object Window]
  ```

- When `this` is used in an object method: In this case, `this` is assigned or bound to the enclosing object. Note that the enclosing object is the immediate parent if you are nesting the objects:

  ```
  var f = {
    name: "f",
    func: function () {
      return this;
    }
  };
  console.log(f.func());
  //prints -
  //[object Object] {
  //  func: function () {
  //    return this;
  //  },
  //  name: "f"
  //}
  ```

- When there is no context: A function, when invoked without any object, does not get any context. By default, it is bound to the global context. When you use `this` in such a function, it is also bound to the global context.
- When `this` is used in a constructor function: As we saw earlier, when a function is called with a `new` keyword, it acts as a constructor. In the case of a constructor, `this` points to the object being constructed. In the following example, `f()` is used as a constructor (because it's invoked with a `new` keyword) and hence, `this` is pointing to the new object being created. So when we say `this.member = "f"`, the new member is added to the object being created, in this case, that object happens to be `o`:

```
var member = "global";
function f()
{
  this.member = "f";
}
var o= new f();
console.log(o.member); // f
```

We saw that instance properties take precedence when the same property is defined both as an instance property and prototype property. It is easy to visualize that when a new object is created, the properties of the constructor's prototype are copied over. However, this is not a correct assumption. What actually happens is that the prototype is attached to the object and referred when any property of this object is referred. Essentially, when a property is referenced on an object, either of the following occur:

- The object is checked for the existence of the property. If it's found, the property is returned.
- The associated prototype is checked. If the property is found, it is returned; otherwise, an `undefined` error is returned.

This is an important understanding because, in JavaScript, the following code actually works perfectly:

```
function Player() {
  isAvailable=false;
}
var crazyBob = new Player();
Player.prototype.isAvailable = function() {
  return isAvailable;
};
console.log(crazyBob.isAvailable()); //false
```

This code is a slight variation of the earlier example. We are creating the object first and then attaching the function to its prototype. When you eventually call the `isAvailable()` method on the object, JavaScript goes to its prototype to search for it if it's not found in the particular object (`crazyBob`, in this case). Think of this as *hot code loading*—when used properly, this ability can give you incredible power to extend the basic object framework even after the object is created.

If you are familiar with OOP already, you must be wondering whether we can control the visibility and access of the members of an object. As we discussed earlier, JavaScript does not have classes. In programming languages such as Java, you have access modifiers such as `private` and `public` that let you control the visibility of the class members. In JavaScript, we can achieve something similar using the function scope as follows:

- You can declare private variables using the `var` keyword in a function. They can be accessed by private functions or privileged methods.
- Private functions may be declared in an object's constructor and can be called by privileged methods.
- Privileged methods can be declared with `this.method=function() {}`.
- Public methods are declared with `Class.prototype.method=function() {}`.
- Public properties can be declared with `this.property` and accessed from outside the object.

The following example shows several ways of doing this:

```
function Player(name,sport,age,country){

  this.constructor.noOfPlayers++;

  // Private Properties and Functions
  // Can only be viewed, edited or invoked by privileged members
  var retirementAge = 40;
  var available=true;
  var playerAge = age?age:18;
  function isAvailable(){ return available &&
(playerAge<retirementAge); }
  var playerName=name ? name :"Unknown";
  var playerSport = sport ? sport : "Unknown";
```

```
    // Privileged Methods
    // Can be invoked from outside and can access private members
    // Can be replaced with public counterparts
    this.book=function(){
      if (!isAvailable()){
        this.available=false;
      } else {
        console.log("Player is unavailable");
      }
    };
    this.getSport=function(){ return playerSport; };
    // Public properties, modifiable from anywhere
    this.batPreference="Lefty";
    this.hasCelebGirlfriend=false;
    this.endorses="Super Brand";
  }

  // Public methods - can be read or written by anyone
  // Can only access public and prototype properties
  Player.prototype.switchHands = function(){ this.
  batPreference="righty"; };
  Player.prototype.dateCeleb = function(){ this.hasCelebGirlfriend=true;
  } ;
  Player.prototype.fixEyes = function(){ this.wearGlasses=false; };

  // Prototype Properties - can be read or written by anyone (or
  overridden)
  Player.prototype.wearsGlasses=true;

  // Static Properties - anyone can read or write
  Player.noOfPlayers = 0;


  (function PlayerTest(){
    //New instance of the Player object created.
    var cricketer=new Player("Vivian","Cricket",23,"England");
    var golfer =new Player("Pete","Golf",32,"USA");
    console.log("So far there are " + Player.noOfPlayers + " in the
  guild");
```

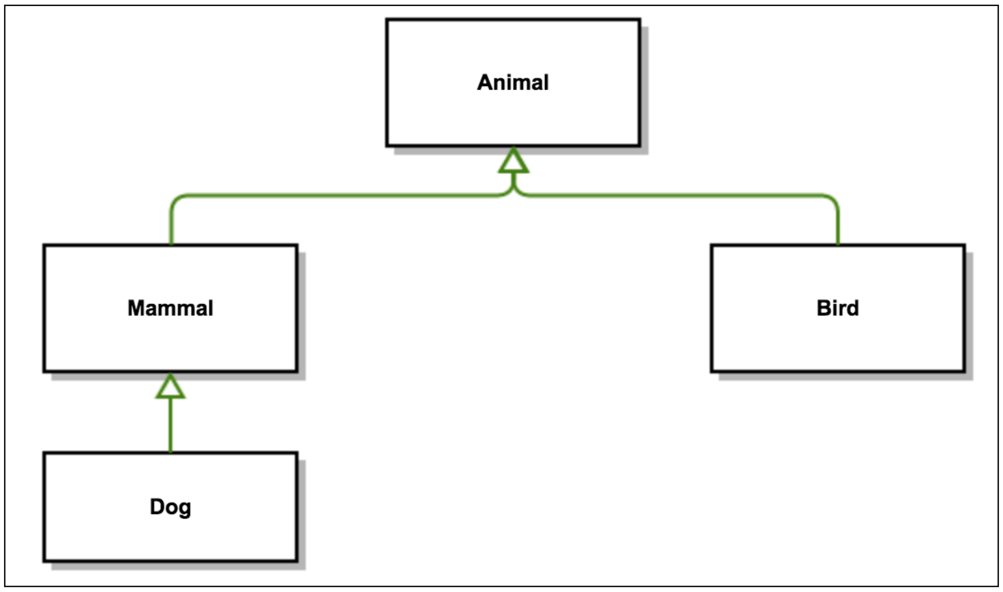
```
    //Both these functions share the common 'Player.prototype.
  wearsGlasses' variable
    cricketer.fixEyes();
    golfer.fixEyes();


    cricketer.endorses="Other Brand";//public variable can be updated

    //Both Player's public method is now changed via their prototype
    Player.prototype.fixEyes=function(){
      this.wearGlasses=true;
    };
    //Only Cricketer's function is changed
    cricketer.switchHands=function(){
      this.batPreference="undecided";
    };

  })();
```

Let's understand a few important concepts from this example:

- The `retirementAge` variable is a private variable that has no privileged method to get or set its value.
- The `country` variable is a private variable created as a constructor argument. Constructor arguments are available as private variables to the object.
- When we called `cricketer.switchHands()`, it was only applied to the `cricketer` and not to both the players, although it's a prototype function of the `Player` itself.
- Private functions and privileged methods are instantiated with each new object created. In our example, new copies of `isAvailable()` and `book()` would be created for each new player instance that we create. On the other hand, only one copy of public methods is created and shared between all instances. This can mean a bit of performance gain. If you don't *really* need to make something private, think about keeping it public.

Inheritance

Inheritance is an important concept of OOP. It is common to have a bunch of objects implementing the same methods. It is also common to have an almost similar object definition with differences in a few methods. Inheritance is very useful in promoting code reuse. We can look at the following classic example of inheritance relation:



Here, you can see that from the generic **Animal** class, we derive more specific classes such as **Mammal** and **Bird** based on specific characteristics. Both the Mammal and Bird classes do have the same template of an Animal; however, they also define behaviors and attributes specific to them. Eventually, we derive a very specific mammal, **Dog**. A Dog has common attributes and behaviors from an Animal class and Mammal class, while it adds specific attributes and behaviors of a Dog. This can go on to add complex inheritance relationships.

Traditionally, inheritance is used to establish or describe an **IS-A** relationship. For example, a dog IS-A mammal. This is what we know as **classical inheritance**. You would have seen such relationships in object-oriented languages such as C++ and Java. JavaScript has a completely different mechanism to handle inheritance. JavaScript is classless language and uses prototypes for inheritance. Prototypal inheritance is very different in nature and needs thorough understanding. Classical and prototypal inheritance are very different in nature and need careful study.

In classical inheritance, instances inherit from a class blueprint and create subclass relationships. You can't invoke instance methods on a class definition itself. You need to create an instance and then invoke methods on this instance. In prototypal inheritance, on the other hand, instances inherit from other instances.

As far as inheritance is concerned, JavaScript uses only objects. As we discussed earlier, each object has a link to another object called its prototype. This prototype object, in turn, has a prototype of its own, and so on until an object is reached with `null` as its prototype; `null`, by definition, has no prototype, and acts as the final link in this prototype chain.

To understand prototype chains better, let's consider the following example:

```
function Person() {}
Person.prototype.cry = function() {
  console.log("Crying");
}
function Child() {}
Child.prototype = {cry: Person.prototype.cry};
var aChild = new Child();
console.log(aChild instanceof Child);  //true
console.log(aChild instanceof Person); //false
console.log(aChild instanceof Object); //true
```

Here, we define a `Person` and then `Child`—a child IS-A person. We also copy the `cry` property of a `Person` to the `cry` property of `Child`. When we try to see this relationship using `instanceof`, we soon realize that just by copying a behavior, we could not really make `Child` an instance of `Person`; `aChild instanceof Person` fails. This is just copying or masquerading, not inheritance. Even if we copy all the properties of `Person` to `Child`, we won't be inheriting from `Person`. This is usually a bad idea and is shown here only for illustrative purposes. We want to derive a prototype chain—an IS-A relationship, a real inheritance where we can say that child IS-A person. We want to create a chain: a child IS-A person IS-A mammal IS-A animal IS-A object. In JavaScript, this is done using an instance of an object as a prototype as follows:

```
SubClass.prototype = new SuperClass();
Child.prototype = new Person();
```

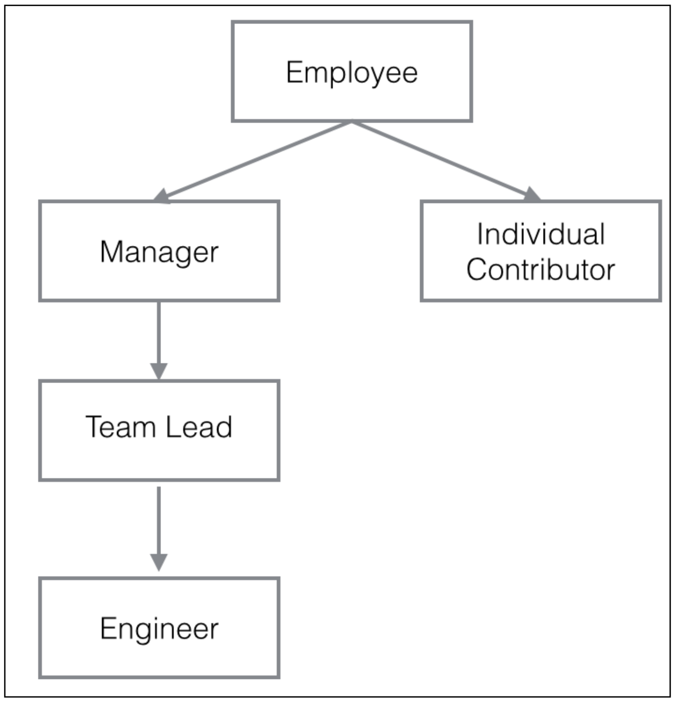
Let's modify the earlier example:

```
function Person() {}
Person.prototype.cry = function() {
  console.log("Crying");
}
```

```
function Child() {}
Child.prototype = new Person();
var aChild = new Child();
console.log(aChild instanceof Child);  //true
console.log(aChild instanceof Person); //true
console.log(aChild instanceof Object); //true
```

The changed line uses an instance of `Person` as the prototype of `Child`. This is an important distinction from the earlier method. Here we are declaring that child IS-A person.

We discussed about how JavaScript looks for a property up the prototype chain till it reaches `Object.prototype`. Let's discuss the concept of prototype chains in detail and try to design the following employee hierarchy:



This is a typical pattern of inheritance. A manager IS-A(n) employee. **Manager** has common properties inherited from an **Employee**. It can have an array of reportees. An **Individual Contributor** is also based on an employee but he does not have any reportees. A **Team Lead** is derived from a Manager with a few functions that are different from a Manager. What we are doing essentially is that each child is deriving properties from its parent (Manager being the parent and Team Lead being the child).

Let's see how we can create this hierarchy in JavaScript. Let's define our `Employee` type:

```
function Employee() {
  this.name = '';
  this.dept = 'None';
  this.salary = 0.00;
}
```

There is nothing special about these definitions. The `Employee` object contains three properties—name, salary, and department. Next, we define `Manager`. This definition shows you how to specify the next object in the inheritance chain:

```
function Manager() {
  Employee.call(this);
  this.reports = [];
}
Manager.prototype = Object.create(Employee.prototype);
```

In JavaScript, you can add a prototypical instance as the value of the prototype property of the constructor function. You can do so at any time after you define the constructor. In this example, there are two ideas that we have not explored earlier. First, we are calling `Employee.call(this)`. If you come from a Java background, this is analogous to the `super()` method call in the constructor. The `call()` method calls a function with a specific object as its context (in this case, it is the given the `this` value), in other words, call allows to specify which object will be referenced by the `this` keyword when the function will be executed. Like `super()` in Java, calling `parentObject.call(this)` is necessary to correctly initialize the object being created.

The other thing we see is `Object.create()` instead of calling `new`. `Object.create()` creates an object with a specified prototype. When we do `new Parent()`, the constructor logic of the parent is called. In most cases, what we want is for `Child.prototype` to be an object that is linked via its prototype to `Parent.prototype`. If the parent constructor contains additional logic specific to the parent, we don't want to run this while creating the child object. This can cause very difficult-to-find bugs. `Object.create()` creates the same prototypal link between the child and parent as the `new` operator without calling the parent constructor.

To have a side effect-free and accurate inheritance mechanism, we have to make sure that we perform the following:

- Setting the prototype to an instance of the parent initializes the prototype chain (inheritance); this is done only once (as the prototype object is shared)
- Calling the parent's constructor initializes the object itself; this is done with every instantiation (you can pass different parameters each time you construct it)

With this understanding in place, let's define the rest of the objects:

```
function IndividualContributor() {
  Employee.call(this);
  this.active_projects = [];
}
IndividualContributor.prototype = Object.create(Employee.prototype);

function TeamLead() {
  Manager.call(this);
  this.dept = "Software";
  this.salary = 100000;
}
TeamLead.prototype = Object.create(Manager.prototype);

function Engineer() {
  TeamLead.call(this);
  this.dept = "JavaScript";
  this.desktop_id = "8822" ;
  this.salary = 80000;
}
Engineer.prototype = Object.create(TeamLead.prototype);
```

Based on this hierarchy, we can instantiate these objects:

```
var genericEmployee = new Employee();
console.log(genericEmployee);
```

You can see the following output for the preceding code snippet:

```
[object Object] {
  dept: "None",
  name: "",
  salary: 0
}
```

A generic `Employee` has a department assigned to `None` (as specified in the default value) and the rest of the properties are also assigned as the default ones.

Next, we instantiate a manager; we can provide specific values as follows:

```
var karen = new Manager();
karen.name = "Karen";
karen.reports = [1,2,3];
console.log(karen);
```

You will see the following output:

```
[object Object] {
  dept: "None",
  name: "Karen",
  reports: [1, 2, 3],
  salary: 0
}
```

For `TeamLead`, the `reports` property is derived from the base class (Manager in this case):

```
var jason = new TeamLead();
jason.name = "Json";
console.log(jason);
```

You will see the following output:

```
[object Object] {
  dept: "Software",
  name: "Json",
  reports: [],
  salary: 100000
}
```

When JavaScript processes the new operator, it creates a new object and passes this object as the value of `this` to the parent—the `TeamLead` constructor. The constructor function sets the value of the `projects` property and implicitly sets the value of the internal `__proto__` property to the value of `TeamLead.prototype`. The `__proto__` property determines the prototype chain used to return property values. This process does not set values for properties inherited from the prototype chain in the `jason` object. When the value of a property is read, JavaScript first checks to see whether the value exists in that object. If the value does exist, this value is returned. If the value is not there, JavaScript checks the prototype chain using the `__proto__` property. Having said this, what happens when you do the following:

```
Employee.prototype.name = "Undefined";
```

It does not propagate to all the instances of `Employee`. This is because when you create an instance of the `Employee` object, this instance gets a local value for the name. When you set the `TeamLead` prototype by creating a new `Employee` object, `TeamLead.prototype` has a local value for the `name` property. Therefore, when JavaScript looks up the `name` property of the `jason` object, which is an instance of `TeamLead`), it finds the local value for this property in `TeamLead.prototype`. It does not try to do further lookups up the chain to `Employee.prototype`.

If you want the value of a property changed at runtime and have the new value be inherited by all the descendants of the object, you cannot define the property in the object's constructor function. To achieve this, you need to add it to the constructor's prototype. For example, let's revisit the earlier example but with a slight change:

```
function Employee() {
  this.dept = 'None';
  this.salary = 0.00;
}
Employee.prototype.name = '';
function Manager() {
  this.reports = [];
}
Manager.prototype = new Employee();
var sandy = new Manager();
var karen = new Manager();

Employee.prototype.name = "Junk";

console.log(sandy.name);
console.log(karen.name);
```

You will see that the `name` property of both sandy and karen has changed to `Junk`. This is because the `name` property is declared outside the constructor function. So, when you change the value of `name` in the Employee's prototype, it propagates to all the descendants. In this example, we are modifying Employee's prototype after the `sandy` and `karen` objects are created. If you changed the prototype before the `sandy` and `karen` objects were created, the value would still have changed to `Junk`.

All native JavaScript objects—Object, Array, String, Number, RegExp, and Function—have prototype properties that can be extended. This means that we can extend the functionality of the language itself. For example, the following snippet extends the `String` object to add a `reverse()` method to reverse a string. This method does not exist in the native String object but by manipulating String's prototype, we add this method to String:

```
String.prototype.reverse = function() {
  return Array.prototype.reverse.apply(this.split('')).join('');
};
var str = 'JavaScript';
console.log(str.reverse()); //"tpircSavaJ"
```

Though this is a very powerful technique, care should be taken not to overuse it. Refer to `http://perfectionkills.com/extending-native-builtins/` to understand the pitfalls of extending native built-ins and what care should be taken if you intend to do so.

Getters and setters

Getters are convenient methods to get the value of specific properties; as the name suggests, **setters** are methods that set the value of a property. Often, you may want to derive a value based on some other values. Traditionally, getters and setters used to be functions such as the following:

```
var person = {
  firstname: "Albert",
  lastname: "Einstein",
  setLastName: function(_lastname){
    this.lastname= _lastname;
  },
  setFirstName: function (_firstname){
    this.firstname= _firstname;
  },
  getFullName: function (){
    return this.firstname + ' '+ this.lastname;
  }
};
person.setLastName('Newton');
person.setFirstName('Issac');
console.log(person.getFullName());
```

As you can see, `setLastName()`, `setFirstName()`, and `getFullName()` are functions used to do *get* and *set* of properties. `Fullname` is a derived property by concatenating the `firstname` and `lastname` properties. This is a very common use case and ECMAScript 5 now provides you with a default syntax for getters and setters.

The following example shows you how getters and setters are created using the object literal syntax in ECMAScript 5:

```
var person = {
  firstname: "Albert",
  lastname: "Einstein",
  get fullname() {
    return this.firstname +" "+this.lastname;
  },
  set fullname(_name){
    var words = _name.toString().split(' ');
    this.firstname = words[0];
    this.lastname = words[1];
  }
};
person.fullname = "Issac Newton";
console.log(person.firstname); //"Issac"
console.log(person.lastname);  //"Newton"
console.log(person.fullname);  //"Issac Newton"
```

Another way of declaring getters and setters is using the `Object.defineProperty()` method:

```
var person = {
  firstname: "Albert",
  lastname: "Einstein",
};
Object.defineProperty(person, 'fullname', {
  get: function() {
    return this.firstname + ' ' + this.lastname;
  },
  set: function(name) {
    var words = name.split(' ');
    this.firstname = words[0];
    this.lastname = words[1];
  }
});
person.fullname = "Issac Newton";
console.log(person.firstname); //"Issac"
console.log(person.lastname);  //"Newton"
console.log(person.fullname);  //"Issac Newton"
```

In this method, you can call `Object.defineProperty()` even after the object is created.

Now that you have tasted the object-oriented flavor of JavaScript, we will go through a bunch of very useful utility methods provided by **Underscore.js**. We discussed the installation and basic usage of Underscore.js in the previous chapter. These methods will make common operations on objects very easy:

- `keys()`: This method retrieves the names of an object's own enumerable properties. Note that this function does not traverse up the prototype chain:

  ```
  var _ = require('underscore');
  var testobj = {
    name: 'Albert',
    age : 90,
    profession: 'Physicist'
  };
  console.log(_.keys(testobj));
  //[ 'name', 'age', 'profession' ]
  ```

- `allKeys()`: This method retrieves the names of an object's own and inherited properties:

  ```
  var _ = require('underscore');
  function Scientist() {
    this.name = 'Albert';
  }
  Scientist.prototype.married = true;
  aScientist = new Scientist();
  console.log(_.keys(aScientist)); //[ 'name' ]
  console.log(_.allKeys(aScientist));//[ 'name', 'married' ]
  ```

- `values()`: This method retrieves the values of an object's own properties:

  ```
  var _ = require('underscore');
  function Scientist() {
    this.name = 'Albert';
  }
  Scientist.prototype.married = true;
  aScientist = new Scientist();
  console.log(_.values(aScientist)); //[ 'Albert' ]
  ```

- `mapObject()`: This method transforms the value of each property in the object:

  ```
  var _ = require('underscore');
  function Scientist() {
    this.name = 'Albert';
    this.age = 90;
  }
  ```

```
aScientist = new Scientist();
var lst = _.mapObject(aScientist, function(val,key){
  if(key==="age"){
    return val + 10;
  } else {
    return val;
  }
});
console.log(lst); //{ name: 'Albert', age: 100 }
```

- `functions()`: This returns a sorted list of the names of every method in an object—the name of every function property of the object.
- `pick()`: This function returns a copy of the object, filtered to just the values of the keys provided:

```
var _ = require('underscore');
var testobj = {
  name: 'Albert',
  age : 90,
  profession: 'Physicist'
};
console.log(_.pick(testobj, 'name','age')); //{ name: 'Albert',
age: 90 }
console.log(_.pick(testobj, function(val,key,object){
  return _.isNumber(val);
})); //{ age: 90 }
```

- `omit()`: This function is an invert of `pick()`—it returns a copy of the object, filtered to omit the values for the specified keys.

Summary

JavaScript applications can improve in clarity and quality by allowing for the greater degree of control and structure that object-orientation can bring to the code. JavaScript object-orientation is based on the function prototypes and prototypal inheritance. These two ideas can provide an incredible amount of wealth to developers.

In this chapter, we saw basic object creation and manipulation. We looked at how constructor functions are used to create objects. We dived into prototype chains and how inheritance operates on the idea of prototype chains. These foundations will be used to build your knowledge of JavaScript patterns that we will explore in the next chapter.

5
JavaScript Patterns

So far, we have looked at several fundamental building blocks necessary to write code in JavaScript. Once you start building larger systems using these fundamental constructs, you soon realize that there can be a standard way of doing a few things. While developing a large system, you will encounter repetitive problems; a pattern intends to provide a standardized solution to such known and identified problems. A pattern can be seen as a best practice, useful abstraction, or template to solve common problems. Writing maintainable code is difficult. The key to write modular, correct, and maintainable code is the ability to understand the repeating themes and use common templates to write optimized solutions to these. The most important text on design patterns was a book published in 1995 called *Design Patterns: Elements Of Reusable Object-Oriented Software* written by *Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides*—a group that became known as the **Gang of Four** (**GOF** for short). This seminal work gave a formal definition to various patterns and explained implementation details of most of the popular patterns that we use today. It is important to understand why patterns are important:

- Patterns offer proven solutions to common problems: Patterns provide templates that are optimized to solve a particular problem. These patterns are backed by solid engineering experience and tested for validity.
- Patterns are designed to be reused: They are generic enough to fit variations of a problem.
- Patterns define vocabulary: Patterns are well-defined structures and hence provide a generic vocabulary to the solution. This can be very expressive when communicating across a larger group.

Design patterns

In this chapter, we will take a look at some of the design patterns that make sense for JavaScript. However, coding patterns are very specific for JavaScript and are of great interest to us. While we spend a lot of time and effort trying to understand and master design patterns, it is important to understand anti-patterns and how to avoid pitfalls. In the usual software development cycle, there are several places where bad code is introduced, mainly around the time where the code is nearing a release or when the code is handed over to a different team for maintenance. If such bad design constructs are documented as anti-patterns, they can provide guidance to developers in knowing what pitfalls to avoid and how not to subscribe to bad design patterns. Most languages have their set of anti-patterns. Based on the kind of problems that they solve, design patterns were categorized into a few broad categories by the GOF:

- **Creational design patterns**: These patterns deal with various mechanisms of object creation. While most languages provide basic object creation methods, these patterns look at optimized or more controlled mechanisms of object creation.
- **Structural design patterns**: These patterns are all about the composition of objects and relationships among them. The idea is to have minimal impact on overall object relationships when something in the system changes.
- **Behavioral design patterns**: These patterns focus on the interdependency and communication between objects.

The following table is a useful ready reckoner to identify categories of patterns:

- Creational patterns:
  - Factory method
  - Abstract factory
  - Builder
  - Prototype
  - Singleton
- Structural patterns:
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade

    - Flyweight
    - Proxy
- Behavioral patterns
    - Interpreter
    - Template method
    - Chain of responsibility
    - Command
    - Iterator
    - Mediator
    - Memento
    - Observer
    - State
    - Strategy
    - Visitor

Some patterns that we will discuss in this chapter may not be part of this list as they are more specific to JavaScript or a variation of these classical patterns. Similarly, we will not discuss patterns that do not fit into JavaScript or are not in popular use.

The namespace pattern

Excessive use of the global scope is almost a taboo in JavaScript. When you build larger programs, it is sometimes difficult to control how much the global scope is polluted. Namespace can reduce the number of globals created by the program and also helps in avoiding naming collisions or excessive name prefixing. The idea of using namespaces is creating a global object for your application or library and adding all these objects and functions to that object rather than polluting the global scope with objects. JavaScript doesn't have an explicit syntax for namespaces, but namespaces can be easily created. Let's consider the following example:

```
function Car() {}
function BMW() {}
var engines = 1;
var features = {
  seats: 6,
  airbags:6
};
```

We are creating all this in the global scope. This is an anti-pattern, and this is never a good idea. We can, however, refactor this code and create a single global object and make all the functions and objects part of this global object as follows:

```
// Single global object
var CARFACTORY = CARFACTORY || {};
CARFACTORY.Car = function () {};
CARFACTORY.BMW = function () {};
CARFACTORY.engines = 1;
CARFACTORY.features = {
  seats: 6,
  airbags:6
};
```

By convention, the global namespace object name is generally written in all caps. This pattern adds namespace to the application and prevents naming collisions in your code and between your code and external library that you use. Many projects use a distinct name after their company or project to create a unique name for their namespace.

Though this seems like an ideal way to restrict your globals and add a namespace to your code, it is a bit verbose; you need to prefix every variable and function with the namespace. You need to type more and the code becomes unnecessarily verbose. Additionally, a single global instance would mean that any part of the code can modify the global instance and the rest of the functionality gets the updated state—this can cause very nasty side-effects. A curious thing to observe in the earlier example is this line—`var CARFACTORY = CARFACTORY || {};`. When you are working on a large code base, you can't assume that you are creating this namespace (or assigning a property to it) for the first time. It is possible that the namespace may pre-exist. To make sure that you create the namespace only if it is not already created, it is safe to always rely on the quick defaulting via a short-circuit `||` operator.

The module pattern

As you build large applications, you will soon realize that it becomes increasingly difficult to keep the code base organized and modular. The module patterns help in keeping the code cleanly separated and organized.

Module separates bigger programs into smaller pieces and gives them a namespace. This is very important because once you separate code into modules, these modules can be reused in several places. Carefully designing interfaces for the modules will make your code very easy to reuse and extend.

JavaScript offers flexible functions and objects that make it easy to create robust module systems. Function scopes help create namespaces that are internal for the module, and objects can be used to store sets of exported values.

Before we start exploring the pattern itself, let's quickly brush up on a few concepts that we discussed earlier.

We discussed object literals in detail. Object literals allow you to create name-value pairs as follows:

```
var basicServerConfig = {
  environment: "production",
  startupParams: {
    cacheTimeout: 30,
    locale: "en_US"
  },
  init: function () {
    console.log( "Initializing the server" );
  },
  updateStartup: function( params ) {
      this.startupParams = params;
      console.log( this.startupParams.cacheTimeout );
      console.log( this.startupParams.locale );
  }
};
basicServerConfig.init(); //"Initializing the server"
basicServerConfig.updateStartup({cacheTimeout:60,
  locale:"en_UK"}); //60, en_UK
```

In this example, we are creating an object literal and defining key-value pairs to create properties and functions.

In JavaScript, the module pattern is used very heavily. Modules help in mimicking the concept of classes. Modules allow us to include both public/private methods and variables of an object, but most importantly, modules restrict these parts from the global scope. As the variables and functions are contained in the module scope, we automatically prevent naming conflict with other scripts using the same names.

Another beautiful aspect of the module pattern is that we expose only a public API. Everything else related to the internal implementation is held private within the module's closure.

Unlike other OO languages, JavaScript has no explicit access modifiers and, hence, there is no concept of *privacy*. You can't have public or private variables. As we discussed earlier, in JavaScript, the function scope can be used to enforce this concept. The module pattern uses closures to restrict variable and function access only within the module; however, variables and functions are defined in the object being returned, which is available to the public.

Let's consider the earlier example and turn this into a module. We are essentially using an IIFE and returning the interface of the module, namely, the `init` and `updateStartup` functions:

```
var basicServerConfig = (function () {
  var environment= "production";
  startupParams= {
    cacheTimeout: 30,
    locale: "en_US"
  };
  return {
    init: function () {
      console.log( "Initializing the server" );
    },
    updateStartup: function( params ) {
      this.startupParams = params;
      console.log( this.startupParams.cacheTimeout );
      console.log( this.startupParams.locale );
    }
  };
})();
basicServerConfig.init(); //"Initializing the server"
basicServerConfig.updateStartup({cacheTimeout:60,
  locale:"en_UK"}); //60, en_UK
```

In this example, `basicServerConfig` is created as a module in the global context. To make sure that we are not polluting the global context with modules, it is important to create namespaces for the modules. Moreover, as modules are inherently reused, it is important to make sure that we avoid naming conflicts using namespaces. For the `basicServerConfig` module, the following snippet shows you the way to create a namespace:

```
// Single global object
var SERVER = SERVER||{};
SERVER.basicServerConfig = (function () {
  Var environment= "production";
  startupParams= {
    cacheTimeout: 30,
    locale: "en_US"
  };
  return {
    init: function () {
      console.log( "Initializing the server" );
    },
    updateStartup: function( params ) {
      this.startupParams = params;
      console.log( this.startupParams.cacheTimeout );
      console.log( this.startupParams.locale );
    }
  };
})();
SERVER.basicServerConfig.init(); //"Initializing the server"
SERVER.basicServerConfig.updateStartup({cacheTimeout:60,
  locale:"en_UK"}); //60, en_UK
```

Using namespace with modules is generally a good idea; however, it is not required that a module must have a namespace associated.

A variation of the module pattern tries to overcome a few problems of the original module pattern. This improved variation of the module pattern is also known as the **revealing module pattern** (**RMP**). RMP was first popularized by *Christian Heilmann*. He disliked that it was necessary to use the module name while calling a public function from another function or accessing a public variable. Another small problem is that you have to use an object literal notation while returning the public interface. Consider the following example:

```
var modulePattern = function(){
  var privateOne = 1;
  function privateFn(){
    console.log('privateFn called');
  }
  return {
    publicTwo: 2,
    publicFn:function(){
      modulePattern.publicFnTwo();
    },
    publicFnTwo:function(){
      privateFn();
    }
  }
}();
modulePattern.publicFn(); "privateFn called"
```

You can see that we need to call `publicFnTwo()` via `modulePattern` in `publicFn()`. Additionally, the public interface is returned in an object literal. The improvement on the classic module pattern is what is known as the RMP. The primary idea behind this pattern is to define all of the members in the private scope and return an anonymous object with pointers to the private functionality that needs to be revealed as public.

Let's see how we can convert our previous example to an RMP. This example is heavily inspired from Christian's blog:

```
var revealingExample = function(){
  var privateOne = 1;
  function privateFn(){
    console.log('privateFn called');
  }
  var publicTwo = 2;
  function publicFn(){
    publicFnTwo();
  }
  function publicFnTwo(){
    privateFn();
  }
  function getCurrentState(){
    return 2;
  }
  // reveal private variables by assigning public pointers
  return {
    setup:publicFn,
    count:publicTwo,
    increaseCount:publicFnTwo,
    current:getCurrentState()
  };
}();
console.log(revealingExample.current); // 2
revealingExample.setup(); //privateFn called
```

An important distinction here is that you define functions and variables in the private scope and return an anonymous object with pointers to the private variables and functions that you want to reveal as public. This is a much cleaner variation and should be preferred over the classic module pattern.

In production code, however, you would want to use more a standardized approach to create modules. Currently, there are two main approaches to create modules. The first is known as **CommonJS modules**. CommonJS modules are usually more suited for server-side JavaScript environments such as **Node.js**. A CommonJS module contains a `require()` function that receives the name of the module and returns the module's interface. The format was proposed by the volunteer group of CommonJS; their aim was to design, prototype, and standardize JavaScript APIs. CommonJS modules consist of two parts. Firstly, list of variables and functions the module needs to expose; when you assign a variable or function to the `module.exports` variable, it is exposed from the module. Secondly, a `require` function that modules can use to import the exports of other modules:

```
//Add a dependency module
var crypto = require('crypto');
function randomString(length, chars) {
  var randomBytes = crypto.randomBytes(length);
  ...
  ...
}
//Export this module to be available for other modules
module.exports=randomString;
```

CommonJS modules are supported by Node.js on the server and **curl.js** in the browser.

The other flavor of JavaScript modules is called **Asynchronous Module Definition** (**AMD**). They are browser-first modules and opt for asynchronous behavior. AMD uses a `define` function to define the modules. This function takes an array of module names and a function. Once the modules are loaded, the `define` function executes the function with their interface as an argument. The AMD proposal is aimed at the asynchronous loading of both the module and dependencies. The `define` function is used to define named or unnamed modules based on the following signature:

```
define(
  module_id /*optional*/,
  [dependencies] /*optional*/,
  definition function /*function for instantiating the module or
    object*/
);
```

You can add a module without dependencies as follows:

```
define(
{
  add: function(x, y){
    return x + y;
  }
});
```

The following snippet shows you a module that depends on two other modules:

```
define( "math",
  //dependency on these two modules
  ["sum", "multiply"],
  // module definition function
  // dependencies (foo and bar) are mapped to function parameters
  function ( sum, multiply ) {
    // return a value that defines the module export
    // (that is, the functionality we want to expose for consumption)

    // create your module here
    var math = {
      demo : function () {
        console.log(sum.calculate(1,2));
        console.log(multiply.calculate(1,2));
      }
    };
  return math;
});
```

The `require` module is used as follows:

```
require(["math","draw"], function ( math,draw ) {
  draw.2DRender(math.pi);
});
```

RequireJS (`http://requirejs.org/docs/whyamd.html`) is one of the module loaders that implements AMD.

ES6 modules

Two separate module systems and different module loaders can be a bit intimidating. ES6 tries to solve this. ES6 has a proposed module specification that tries to keep the good aspects of both the CommonJS and AMD module patterns. The syntax of ES6 modules is similar to CommonJS and the ES6 modules support asynchronous loading and configurable module loading:

```
//json_processor.js
function processJSON(url) {
  ...
}
export function getSiteContent(url) {
  return processJSON(url);
}
//main.js
import { getSiteContent } from "json_processor.js";
content=getSiteContent("http://google.com/");
```

ES6 export lets you export a function or variable in a way similar to CommonJS. In the code where you want to use this imported function, you use the `import` keyword to specify from where you want the dependency to be imported. Once the dependency is imported, it can be used as a member of the program. We will discuss in later chapters how you can use ES6 in environments where ES6 is not supported.

The factory pattern

The factory pattern is another popular object creation pattern. It does not require the usage of constructors. This pattern provides an interface to create objects. Based on the type passed to the factory, that particular type of object is created by the factory. A common implementation of this pattern is usually using a class or static method of a class. The purposes of such a class or method are as follows:

- It abstracts out repetitive operations when creating similar objects
- It allows the consumers of the factory to create objects without knowing the internals of the object creation

Let's take a common example to understand the usage of a factory. Let's say that we have the following:

- A constructor, `CarFactory()`
- A static method in `CarFactory` called `make()` that knows how to create objects of the `car` type
- Specific `car` types such as `CarFactory.SUV`, `CarFactory.Sedan`, and so on

We want to use `CarFactory` as follows:

```
var golf = CarFactory.make('Compact');
var vento = CarFactory.make('Sedan');
var touareg = CarFactory.make('SUV');
```

Here is how you would implement such a factory. The following implementation is fairly standard. We are programmatically calling the constructor function that creates an object of the specified type—`CarFactory[const].prototype = new CarFactory();`.

We are mapping object types to the constructors. There can be variations in how you can go about implementing this pattern:

```
// Factory Constructor
function CarFactory() {}
CarFactory.prototype.info = function() {
  console.log("This car has "+this.doors+" doors and a
    "+this.engine_capacity+" liter engine");
};
// the static factory method
CarFactory.make = function (type) {
  var constr = type;
  var car;
  CarFactory[constr].prototype = new CarFactory();
  // create a new instance
  car = new CarFactory[constr]();
  return car;
};

CarFactory.Compact = function () {
  this.doors = 4;
  this.engine_capacity = 2;
};
```

```
CarFactory.Sedan = function () {
  this.doors = 2;
  this.engine_capacity = 2;
};
CarFactory.SUV = function () {
  this.doors = 4;
  this.engine_capacity = 6;
};
  var golf = CarFactory.make('Compact');
  var vento = CarFactory.make('Sedan');
  var touareg = CarFactory.make('SUV');
  golf.info(); //"This car has 4 doors and a 2 liter engine"
```

We suggest that you try this example in JS Bin and understand the concept by actually writing its code.

The mixin pattern

Mixins help in significantly reducing functional repetition in our code and help in function reuse. We can move this shared functionality to a mixin and reduce duplication of shared behavior. You can then focus on building the actual functionality and not keep repeating the shared behavior. Let's consider the following example. We want to create a custom logger that can be used by any object instance. The logger will become a functionality shared across objects that want to use/extend the mixin:

```
var _ = require('underscore');
//Shared functionality encapsulated into a CustomLogger
var logger = (function () {
  var CustomLogger = {
    log: function (message) {
      console.log(message);
    }
  };
  return CustomLogger;
}());

//An object that will need the custom logger to log system
  specific logs
var Server = (function (Logger) {
  var CustomServer = function () {
    this.init = function () {
      this.log("Initializing Server...");
    };
  };

  // This copies/extends the members of the 'CustomLogger' into
    'CustomServer'
  _.extend(CustomServer.prototype, Logger);
  return CustomServer;
}(logger));

(new Server()).init(); //Initializing Server...
```

In this example, we are using `_.extend` from **Underscore.js**—we discussed this function in the previous chapter. This function is used to copy all the properties from the source (`Logger`) to the destination (`CustomServer.prototype`). As you can observe in this example, we are creating a shared `CustomLogger` object that is intended to be used by any object instance needing its functionality. One such object is `CustomServer`—in its `init()` method, we call this custom logger's `log()` method. This method is available to `CustomServer` because we are extending `CustomLogger` via Underscore's `extend()`. We are dynamically adding functionality of a mixin to the consumer object. It is important to understand the distinction between mixins and inheritance. When you have shared functionality across multiple objects and class hierarchies, you can use mixins. If you have shared functionality along a single class hierarchy, you can use inheritance. In prototypical inheritance, when you inherit from a prototype, any change to the prototype affects everything that inherits the prototype. If you do not want this to happen, you can use mixins.

The decorator pattern

The primary idea behind the decorator pattern is that you start your design with a plain object, which has some basic functionality. As the design evolves, you can use existing decorators to enhance your plain object. This is a very popular pattern in the OO world and especially in Java. Let's take an example of `BasicServer`—a server with very basic functionality. This basic functionality can be decorated to serve specific purposes. We can have two different cases where this server can serve both PHP and Node.js and serve them on different ports. These different functionality are decorated to the basic server:

```
var phpServer = new BasicServer();
phpServer = phpServer.decorate('reverseProxy');
phpServer = phpServer.decorate('servePHP');
phpServer = phpServer.decorate('80');
phpServer = phpServer.decorate('serveStaticAssets');
phpServer.init();
```

The Node.js server will have something as follows:

```
var nodeServer = new BasicServer();
nodeServer = nodeServer.decorate('serveNode');
nodeServer = nodeServer.decorate('3000');
nodeServer.init();
```

There are several ways in which the decorator pattern is implemented in JavaScript. We will discuss a method where the pattern is implemented by a list and does not rely on inheritance or method call chain:

```
//Implement BasicServer that does the bare minimum
function BasicServer() {
  this.pid = 1;
  console.log("Initializing basic Server");
  this.decorators_list = []; //Empty list of decorators
}
//List of all decorators
BasicServer.decorators = {};

//Add each decorator to the list of BasicServer's decorators
//Each decorator in this list will be applied on the BasicServer
  instance
BasicServer.decorators.reverseProxy = {
  init: function(pid) {
    console.log("Started Reverse Proxy");
    return pid + 1;
  }
};
BasicServer.decorators.servePHP = {
  init: function(pid) {
    console.log("Started serving PHP");
    return pid + 1;
  }
};
BasicServer.decorators.serveNode = {
  init: function(pid) {
    console.log("Started serving Node");
    return pid + 1;
  }
};
```

```
//Push the decorator to this list everytime decorate() is called
BasicServer.prototype.decorate = function(decorator) {
  this.decorators_list.push(decorator);
};
//init() method looks through all the applied decorators on
  BasicServer
//and executes init() method on all of them
BasicServer.prototype.init = function () {
  var running_processes = 0;
  var pid = this.pid;
  for (i = 0; i < this.decorators_list.length; i += 1) {
    decorator_name = this.decorators_list[i];
    running_processes =
      BasicServer.decorators[decorator_name].init(pid);
  }
  return running_processes;
};

//Create server to serve PHP
var phpServer = new BasicServer();
phpServer.decorate('reverseProxy');
phpServer.decorate('servePHP');
total_processes = phpServer.init();
console.log(total_processes);

//Create server to serve Node
var nodeServer = new BasicServer();
nodeServer.decorate('serveNode');
nodeServer.init();
total_processes = phpServer.init();
console.log(total_processes);
```

`BasicServer.decorate()` and `BasicServer.init()` are the two methods where the real stuff happens. We push all decorators being applied to the list of decorators for `BasicServer`. In the `init()` method, we execute or apply each decorator's `init()` method from this list of decorators. This is a cleaner approach to decorator patterns that does not use inheritance. This method was described by Stoyan Stefanov in his book, *JavaScript Patterns, O'Reilly Media*, and has gained prominence among JavaScript developers due to its simplicity.

The observer pattern

Let's first see the language-agnostic definition of the observer pattern. The GOF book, *Design Patterns: Elements of Reusable Object-Oriented Software*, defines the observer pattern as follows:

One or more observers are interested in the state of a subject and register their interest with the subject by attaching themselves. When something changes in our subject that the observer may be interested in, a notify message is sent which calls the update method in each observer. When the observer is no longer interested in the subject's state, they can simply detach themselves.

In the observer design pattern, the subject keeps a list of objects depending on it (called observers) and notifies them when the state changes. The subject uses a broadcast to the observers to inform them of the change. Observers can remove themselves from the list once they no longer wish to be notified. Based on this understanding, we can define the participants in this pattern:

- **Subject**: This keeps the list of observers and has methods to add, remove, and update observers
- **Observer**: This provides an interface for objects that need to be notified when the subject's state changes

Let's create a subject that can add, remove, and notify observers:

```
var Subject = ( function(  ) {
  function Subject() {
    this.observer_list = [];
  }
  // this method will handle adding observers to the internal list
  Subject.prototype.add_observer = function ( obj ) {
    console.log( 'Added observer' );
    this.observer_list.push( obj );
  };
  Subject.prototype.remove_observer = function ( obj ) {
    for( var i = 0; i < this.observer_list.length; i++ ) {
      if( this.observer_list[ i ] === obj ) {
        this.observer_list.splice( i, 1 );
        console.log( 'Removed Observer' );
      }
    }
  };
  Subject.prototype.notify = function () {
    var args = Array.prototype.slice.call( arguments, 0 );
    for( var i = 0; i<this.observer_list.length; i++ ) {
      this.observer_list[i].update(args);
    }
  };
  return Subject;
})();
```

This is a fairly straightforward implementation of a `Subject`. The important fact about the `notify()` method is the way in which all the observer objects' `update()` methods are called to broadcast the update.

Now let's define a simple object that creates random tweets. This object is providing an interface to add and remove observers to the `Subject` via `addObserver()` and `removeObserver()` methods. It also calls the `notify()` method of `Subject` with the newly fetched tweet. When this happens, all the observers will broadcast that the new tweet has been updated with the new tweet being passed as the parameter:

```
function Tweeter() {
  var subject = new Subject();
  this.addObserver = function ( observer ) {
    subject.add_observer( observer );
  };
  this.removeObserver = function (observer) {
    subject.remove_observer(observer);
  };
  this.fetchTweets = function fetchTweets() {
    // tweet
    var tweet = {
      tweet: "This is one nice observer"
    };
    // notify our observers of the stock change
    subject.notify( tweet );
  };
}
```

Let's now add two observers:

```
var TweetUpdater = {
  update : function() {
    console.log( 'Updated Tweet -  ', arguments );
  }
};
```

```
var TweetFollower = {
  update : function() {
    console.log( '"Following this tweet -  ', arguments );
  }
};
```

Both these observers will have one `update()` method that will be called by the `Subject.notify()` method. Now we can actually add these observers to the `Subject` via Tweeter's interface:

```
var tweetApp = new Tweeter();
tweetApp.addObserver( TweetUpdater );
tweetApp.addObserver( TweetFollower );
tweetApp.fetchTweets();
tweetApp.removeObserver(TweetUpdater);
tweetApp.removeObserver(TweetFollower);
```

This will result in the following output:

```
Added observer
Added observer
Updated Tweet -   { '0': [ { tweet: 'This is one nice observer' } ] }
"Following this tweet -   { '0': [ { tweet: 'This is one nice
observer' } ] }
Removed Observer
Removed Observer
```

This is a basic implementation to illustrate the idea of the observer pattern.

JavaScript Model-View-* patterns

Model-View-Controller (**MVC**), **Model-View-Presenter** (**MVP**), and **Model-View-ViewModel** (**MVVM**) have been popular with server applications, but in recent years JavaScript applications are also using these patterns to structure and manage large projects. Many JavaScript frameworks have emerged that support **MV*** patterns. We will discuss a few examples using **Backbone.js**.

Model-View-Controller

MVC is a popular structural pattern where the idea is to divide an application into three parts so as to separate the internal representations of information from the presentation layer. MVC consists of components. The model is the application object, view is the presentation of the underlying model object, and controller handles the way in which the user interface behaves, depending on the user interactions.

Models

Models are constructs that represent data in the applications. They are agnostic of the user interface or routing logic. Changes to models are typically notified to the view layer by following the observer design pattern. Models may also contain code to validate, create, or delete data. The ability to automatically notify the views to react when the data is changed makes frameworks such as Backbone.js, **Amber.js**, and others very useful in building MV* applications. The following example shows you a typical Backbone model:

```
var EmployeeModel = Backbone.Model.extend({
  url: '/employee/1',
  defaults: {
    id: 1,
    name: 'John Doe',
    occupation: null
  }
  initialize: function() {
 }
}); var JohnDoe = new EmployeeModel();
```

This model structure may vary between different frameworks but they usually have certain commonalities in them. In most real-world applications, you would want your model to be persisted to an in-memory store or database.

Views

Views are the visual representations of your model. Usually, the state of the model is processed, filtered, or massaged before it is presented to the view layer. In JavaScript, views are responsible for rendering and manipulating DOM elements. Views observe models and get notified when there is a change in the model. When the user interacts with the view, certain attributes of the model are changed via the view layer (usually via controllers). In JavaScript frameworks such as Backbone, the views are created using template engines such as **Handlebar.js** (`http://handlebarsjs.com/`) or **mustache.js** (`https://mustache.github.io/`). These templates themselves are not views. They observe models and keep the view state updated based on these changes. Let's see an example of a view defined in Handlebar:

```
<li class="employee_photo">
  <h2>{{title}}</h2>
  <img class="emp_headshot_small" src="{{src}}"/>
  <div class="employee_details">
    {{employee_details}}
  </div>
</li>
```

Views such as the preceding example contain markup tags containing template variables. These variables are delimited via a custom syntax. For example, template variables are delimited using `{{ }}` in Handlebar.js. Frameworks typically transmit data in JSON format. How the view is populated from the model is handled transparently by the framework.

Controllers

Controllers act as a layer between models and views and are responsible for updating the model when the user changes the view attributes. Most JavaScript frameworks deviate from the classical definition of a controller. For example, Backbone does not have a concept called controller; they have something called a **router** that is responsible to handle routing logic. You can think of a combination of the view and router as a controller because a lot of the logic to synchronize models and views is done within the view itself. A typical Backbone router would look as follows:

```
var EmployeeRouter = Backbone.Router.extend({
  routes: { "employee/:id": "route" },
  route: function( id ) {
    ...view render logic...
  }
});
```

The Model-View-Presenter pattern

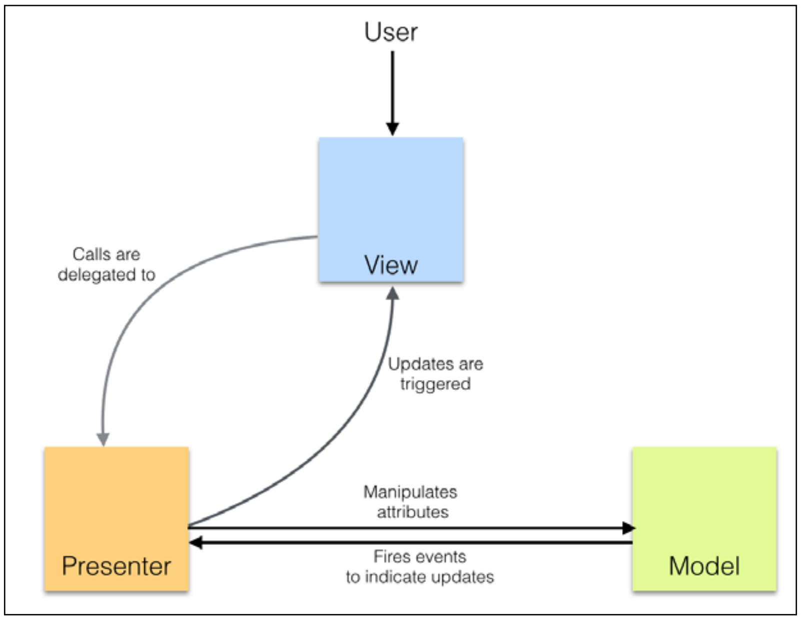
Model-View-Presenter is a variation of the original MVC pattern that we discussed previously. Both MVC and MVP target the separation of concerns but they are different on many fundamental aspects. The presenter in MVP has the necessary logic for the view. Any invocation from the view gets delegated to the presenter. The presenter also observes the model and updates the views when the model updates. Many authors take the view that because the presenter binds the model with views, it also performs the role of a traditional controller. There are various implementations of MVP and there are no frameworks that offer classical MVP out of the box. In implementations of MVP, the following are the primary differences that separate MVP from MVC:

- The view has no reference to the model
- The presenter has a reference to the model and is responsible for updating the view when the model changes

MVP is generally implemented in two flavors:

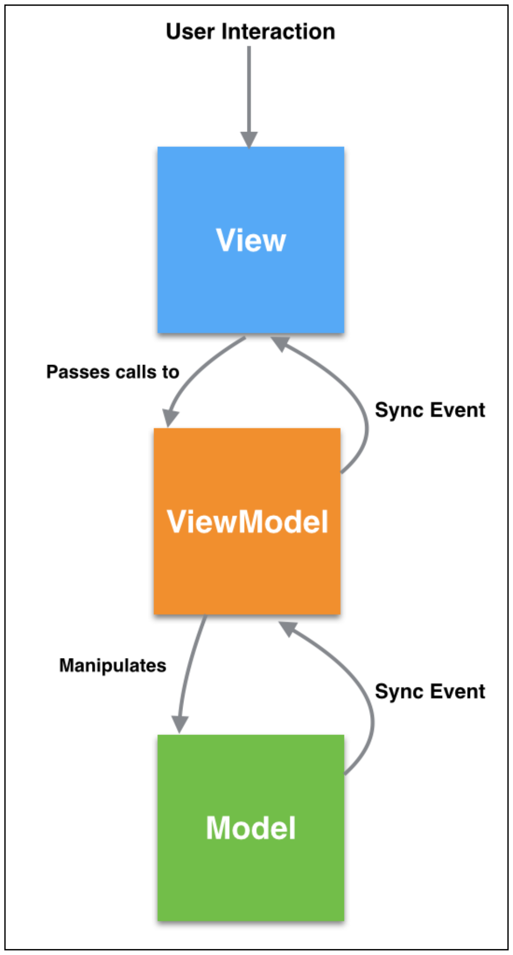
- Passive view: The view is as naïve as possible and all the business logic is within the presenter. For example, a plain Handlebars template can be seen as a passive view.
- Supervising controller: Views mostly contain declarative logic. A presenter takes over when the simple declarative logic in the view is insufficient.

The following figure depicts MVP architecture:



Model-View-ViewModel

MVVM was originally coined by Microsoft for use with **Windows Presentation Foundation** (**WPF**) and **Silverlight**. MVVM is a variation of MVC and MVP and further tries to separate the user interface (view) from the business model and application behavior. MVVM creates a new model layer in addition to the domain model that we discussed in MVC and MVP. This model layer adds properties as an interface for the view. Let's say that we have a checkbox on the UI. The state of the checkbox is captured in an `IsChecked` property. In MVP, the view will have this property and the presenter will set it. However, in MVVM, the presenter will have the `IsChecked` property and the view is responsible for syncing with it. Now that the presenter is not really doing the job of a classical presenter, it's renamed as ViewModel:



Implementation details of these approaches are dependent on the problem that we are trying to solve and the framework that we use.

Summary

While building large applications, we see certain problem patterns repeating over and over. These patterns have well-defined solutions that can be reused to build a robust solution. In this chapter, we discussed some of the important patterns and ideas around these patterns. Most modern JavaScript applications use these patterns. It is rare to see a large-scale system built without implementing modules, decorators, factories, or MV* patterns. These are foundational ideas that we discussed in this chapter. We will discuss various testing and debugging techniques in the next chapter.

6 Testing and Debugging

As you write JavaScript applications, you will soon realize that having a sound testing strategy is indispensable. In fact, not writing enough tests is almost always a bad idea. It is essential to cover all the non-trivial functionality of your code to make sure of the following points:

- The existing code behaves as per the specifications
- Any new code does not break the behavior defined by the specifications

Both these points are very important. Many engineers consider only the first point the sole reason to cover your code with enough tests. The most obvious advantage of test coverage is to really make sure that the code being pushed to the production system is mostly error-free. Writing test cases to smartly cover the maximum functional areas of the code generally gives you a good indication about the overall quality of the code. There should be no arguments or compromises around this point. It is unfortunate though that many production systems are still bereft of adequate code coverage. It is very important to build an engineering culture where developers think about writing tests as much as they think about writing code.

The second point is even more important. Legacy systems are usually very difficult to manage. When you are working on code written either by someone else or a large distributed team, it is fairly easy to introduce bugs and break things. Even the best engineers make mistakes. When you are working on a large code base that you are unfamiliar with and if there is no sound test coverage to help you, you will introduce bugs. As you won't have the confidence in the changes that you are making (because there are no test cases to confirm your changes), your code releases will be shaky, slow, and obviously full of hidden bugs.

You will refrain from refactoring or optimizing your code because you won't really be sure what changes to the code base would potentially break something (again, because there are no test cases to confirm your changes)—all this is a vicious circle. It's like a civil engineer saying, "though I have constructed this bridge, I have no confidence in the quality of the construction. It may collapse immediately or never." Though this may sound like an exaggeration, I have seen a lot of high impact production code being pushed with no test coverage. This is risky and should be avoided. When you are writing enough test cases to cover majority of your functional code and when you make a change to these pieces, you immediately realize if there is a problem with this new change. If your changes make the test case fail, you realize the problem. If your refactoring breaks the test scenario, you realize the problem—all this happens much before the code is pushed to production.

In recent years, ideas such as test-driven development and self-testing code are gaining prominence, especially in **agile methodology**. These are fundamentally sound ideas and will help you write robust code—code that you are confident of. We will discuss all these ideas in this chapter. You will understand how to write good test cases in modern JavaScript. We will also look at several tools and methods to debug your code. JavaScript has been traditionally a bit difficult to test and debug primarily due to lack of tools, but modern tools make both of these easy and natural.

Unit testing

When we talk about test cases, we mostly mean **unit tests**. It is incorrect to assume that the unit that we want to test is always a function. The unit (or unit of work) is a logical unit that constitutes a single behavior. This unit should be able to be invoked via a public interface and should be testable independently.

Thus, a unit test performs the following functions:

- It tests a single logical function
- It can be run without a specific order of execution
- It takes care of its own dependencies and mock data
- It always returns the same result for the same input
- It should be self-explanatory, maintainable, and readable

Martin Fowler advocates the **test pyramid** (`http://martinfowler.com/bliki/TestPyramid.html`) strategy to make sure that we have a high number of unit tests to ensure maximum code coverage. The test pyramid says that you should write many more low-level unit tests than higher level integration and UI tests.

There are two important testing strategies that we will discuss in this chapter.

Test-driven development

Test-driven development (**TDD**) has gained a lot of prominence in the last few years. The concept was first proposed as part of the **Extreme Programming** methodology. The idea is to have short repetitive development cycles where the focus is on writing the test cases first. The cycle looks as follows:

1. Add a test case as per the specifications for a specific unit of code.
2. Run the existing suite of test cases to see if the new test case that you wrote fails—it should (because there is no code for this unit yet). This step ensures that the current test harness works well.
3. Write the code that serves mainly to confirm the test case. This code is not optimized or refactored or even entirely correct. However, this is fine at the moment.
4. Rerun the tests and see if all the test cases pass. After this step, you will be confident that the new code is not breaking anything.
5. Refactor the code to make sure that you are optimizing the unit and handling all corner cases.

These steps are repeated for all the new code that you add. This is an elegant strategy that works really well for the agile methodology. TDD will be successful only if the testable units of code are small and confirm only to the test case and nothing more. It is important to write small, modular, and precise code units that have input and output confirming the test case.

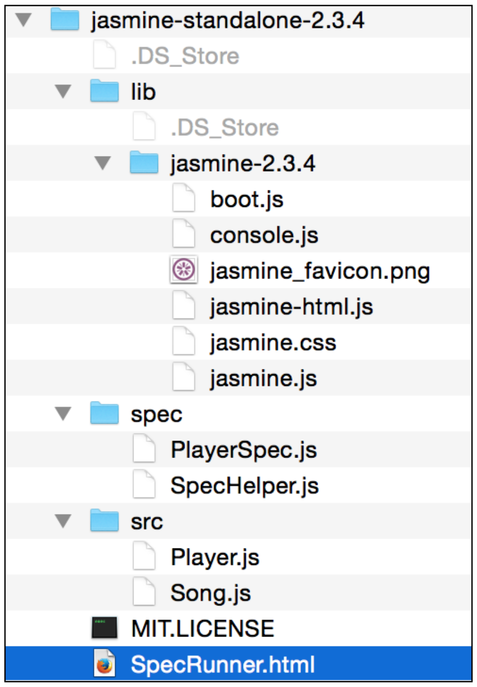
Behavior-driven development

A very common problem while trying to follow TDD is vocabulary and the definition of *correctness*. BDD tries to introduce a *ubiquitous language* while writing the test cases when you are following TDD. This language makes sure that both the business and engineering teams are talking about the same thing.

We will use **Jasmine** as the primary BDD framework and explore various testing strategies.

> You can install Jasmine by downloading the standalone package from `https://github.com/jasmine/jasmine/releases/download/v2.3.4/jasmine-standalone-2.3.4.zip`.

When you unzip this package, you will have the following directory structure:



```
jasmine-standalone-2.3.4
    .DS_Store
    lib
        .DS_Store
        jasmine-2.3.4
            boot.js
            console.js
            jasmine_favicon.png
            jasmine-html.js
            jasmine.css
            jasmine.js
    spec
        PlayerSpec.js
        SpecHelper.js
    src
        Player.js
        Song.js
    MIT.LICENSE
    SpecRunner.html
```

The `lib` directory contains the JavaScript files that you need in your project to start writing Jasmine test cases. If you open `SpecRunner.html`, you will find the following JavaScript files included in it:

```
<script src="lib/jasmine-2.3.4/jasmine.js"></script>
<script src="lib/jasmine-2.3.4/jasmine-html.js"></script>
<script src="lib/jasmine-2.3.4/boot.js"></script>

<!-- include source files here... -->
<script src="src/Player.js"></script>
```

```
<script src="src/Song.js"></script>
<!-- include spec files here... -->
<script src="spec/SpecHelper.js"></script>
<script src="spec/PlayerSpec.js"></script>
```

The first three are Jasmine's own framework files. The next section includes the source files that we want to test and the actual test specifications.

Let's experiment with Jasmine with a very ordinary example. Create a `bigfatjavascriptcode.js` file and place it in the `src/` directory. We will test the following function:

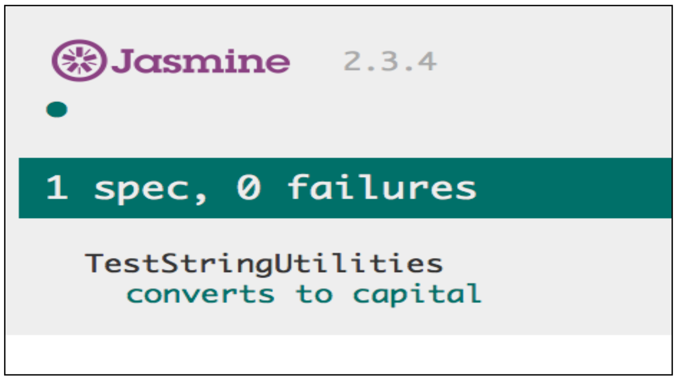
```
function capitalizeName(name){
  return name.toUpperCase();
}
```

This is a simple function that does one single thing. It receives a string and returns a capitalized string. We will test various scenarios around this function. This is the unit of code that we discussed earlier.

Next, create the test specifications. Create one JavaScript file, `test.spec.js`, and place it in the `spec/` directory. The file should contain the following. You will need to add the following two lines to `SpecRunner.html`:

```
<script src="src/bigfatjavascriptcode.js"></script>
<script src="spec/test.spec.js"></script>
```

The order of this inclusion does not matter. When we run `SpecRunner.html`, you will see something as follows:

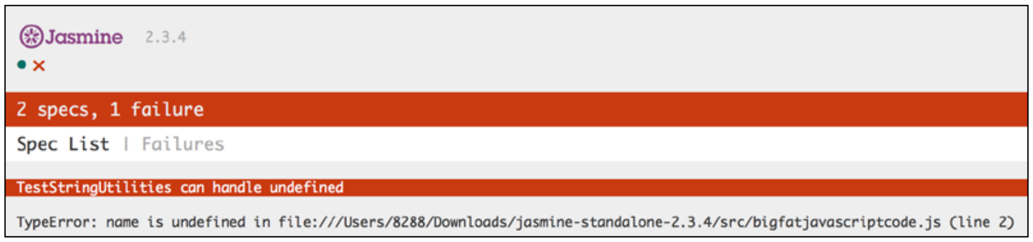


```
1 spec, 0 failures
TestStringUtilities
  converts to capital
```

This is the Jasmine report that shows the details about the number of tests that were executed and the count of failures and successes. Now, let's make the test case fail. We want to test a case where an undefined variable is passed to the function. Add one more test case as follows:

```
it("can handle undefined", function() {
  var str= undefined;
  expect(capitalizeName(str)).toEqual(undefined);
});
```

Now, when you run `SpecRunner.html`, you will see the following result:

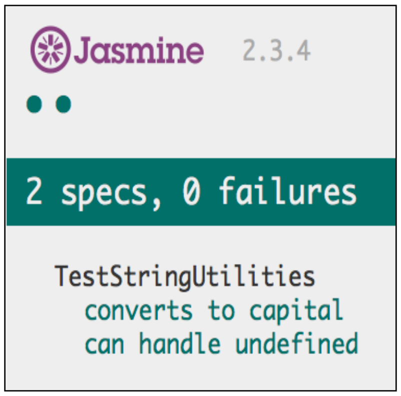


```
Jasmine 2.3.4
• ×
2 specs, 1 failure
Spec List | Failures
TestStringUtilities can handle undefined
TypeError: name is undefined in file:///Users/8288/Downloads/jasmine-standalone-2.3.4/src/bigfatjavascriptcode.js (line 2)
```

As you can see, the failure is displayed for this test case in a detailed error stack. Now, we go about fixing this. In your original JavaScript code, we can handle an undefined condition as follows:

```
function capitalizeName(name){
  if(name){
    return name.toUpperCase();
  }
}
```

With this change, your test case will pass and you will see the following in the Jasmine report:



```
Jasmine 2.3.4
• •
2 specs, 0 failures
TestStringUtilities
  converts to capital
  can handle undefined
```

This is very similar to what a test-driven development would look. You write test cases, you then fill in the necessary code to confirm to the specifications, and rerun the test suite. Let's understand the structure of the Jasmine tests.

Our test specification looks as follows:

```
describe("TestStringUtilities", function() {
  it("converts to capital", function() {
    var str = "albert";
    expect(capitalizeName(str)).toEqual("ALBERT");
  });
  it("can handle undefined", function() {
    var str= undefined;
    expect(capitalizeName(str)).toEqual(undefined);
  });
});
```

The `describe("TestStringUtilities"` is a test suite. The name of the test suite should describe the unit of code that we are testing—this can be a function or group of related functionality. In the specifications, you call the global Jasmine `it` function to which you pass the title of the specification and test function used by the test case. This function is the actual test case. You can catch one or more assertions or the general expectations using the `expect` function. When all expectations are `true`, your specification is passed. You can write any valid JavaScript code in the `describe` and `it` functions. The values that you verify as part of the expectations are matched using a matcher. In our example, `toEqual()` is the matcher that matches two values for equality. Jasmine contains a rich set of matches to suit most of the common use cases. Some common matchers supported by Jasmine are as follows:

- `toBe()`: This matcher checks whether two objects being compared are equal. This is the same as the `===` comparison, as shown in the following code:

  ```
  var a = { value: 1};
  var b = { value: 1 };

  expect(a).toEqual(b);  // success, same as == comparison
  expect(b).toBe(b);     // failure, same as === comparison
  expect(a).toBe(a);     // success, same as === comparison
  ```

- `not`: You can negate a matcher with a `not` prefix. For example, `expect(1).not.toEqual(2);` will negate the match made by `toEqual()`.

- `toContain()`: This checks whether an element is part of an array. This is not an exact object match as `toBe()`. For example, look at the following code:

  ```
  expect([1, 2, 3]).toContain(3);
  expect("astronomy is a science").toContain("science");
  ```

- `toBeDefined()` and `toBeUndefined()`: These two matches are handy to check whether a variable is undefined (or not).
- `toBeNull()`: This checks whether a variable's value is `null`.
- `toBeGreaterThan()` and `toBeLessThan()`: These matchers perform numeric comparisons (they work on strings too):

  ```
  expect(2).toBeGreaterThan(1);
  expect(1).toBeLessThan(2);
  expect("a").toBeLessThan("b");
  ```

One interesting feature of Jasmine is the **spies**. When you are writing a large system, it is not possible to make sure that all systems are always available and correct. At the same time, you don't want your unit tests to fail due to a dependency that may be broken or unavailable. To simulate a situation where all dependencies are available for a unit of code that we want to test, we mock these dependencies to always give the response that we expect. Mocking is an important aspect of testing and most testing frameworks provide support for the mocking. Jasmine allows mocking using a feature called a spy. Jasmine spies essentially stub the functions that we may not have ready; at the time of writing the test case but as part of the functionality, we need to track that we are executing these dependencies and not ignoring them. Consider the following example:

```
describe("mocking configurator", function() {
  var configurator = null;
  var responseJSON = {};

  beforeEach(function() {
    configurator = {
      submitPOSTRequest: function(payload) {
        //This is a mock service that will eventually be replaced
        //by a real service
        console.log(payload);
        return {"status": "200"};
      }
    };
    spyOn(configurator,
      'submitPOSTRequest').and.returnValue({"status": "200"});
    configurator.submitPOSTRequest({
```

```
        "port":"8000",
        "client-encoding":"UTF-8"
      });
    });

    it("the spy was called", function() {
      expect(configurator.submitPOSTRequest).toHaveBeenCalled();
    });

    it("the arguments of the spy's call are tracked", function() {
      expect(configurator.submitPOSTRequest).toHaveBeenCalledWith({"port
  ":"8000","client-encoding":"UTF-8"});
    });
  });
```

In this example, while we are writing this test case, we either don't have the real implementation of the `configurator.submitPOSTRequest()` dependency or someone is fixing this particular dependency. In any case, we don't have it available. For our test to work, we need to mock it. Jasmine spies allow us to replace a function with its mock and track its execution.

In this case, we need to ensure that we called the dependency. When the actual dependency is ready, we will revisit this test case to make sure that it fits the specifications, but at this time, all that we need to ensure is that the dependency is called. The Jasmine `tohaveBeenCalled()` function lets us track the execution of a function, which may be a mock. We can use `toHaveBeenCalledWith()` that allows us to determine if the stub function was called with the correct parameters. There are several other interesting scenarios that you can create using Jasmine spies. The scope of this chapter won't permit us to cover them all, but I would encourage you to discover these areas on your own.

You can refer to the user manual for Jasmine for more information on Jasmine spies at `http://jasmine.github.io/2.0/introduction.html`.

Mocha, Chai, and Sinon

Though Jasmine is the most prominent JavaScript testing framework, **Mocha** and **Chai** are gaining prominence in the Node.js environment. Mocha is the testing framework used to describe and run test cases. Chai is the assertion library supported by Mocha. **Sinon.JS** comes in handy while creating mocks and stubs for your tests. We won't discuss these frameworks in this book, but experience on Jasmine will be handy if you want to experiment with these frameworks.

JavaScript debugging

If you are not a completely new programmer, I am sure you must have spent some amount of time debugging your or someone else's code. Debugging is almost like an art form. Every language has different methods and challenges around debugging. JavaScript has traditionally been a difficult language to debug. I have personally spent days and nights of misery trying to debug badly-written JavaScript code using `alert()` functions. Fortunately, modern browsers such as Mozilla Firefox and Google Chrome have excellent developer tools to help debug JavaScript in the browser. There are IDEs like **IntelliJ WebStorm** with great debugging support for JavaScript and Node.js. In this chapter, we will focus primarily on Google Chrome's built-in developer tool. Firefox also supports the Firebug extension and has excellent built-in developer tools, but as they behave more or less the same as Google Chrome's **Developer Tools** (**DevTools**), we will discuss common debugging approaches that work in both of these tools.

Before we talk about the specific debugging techniques, let's understand the type of errors that we would be interested in while we try to debug our code.

Syntax errors

When your code has something that does not confirm to the JavaScript language grammar, the interpreter rejects this piece of code. These are easy to catch if your IDE is helping you with syntax checking. Most modern IDEs help with these errors. Earlier, we discussed the usefulness of the tools such as **JSLint** and **JSHint** around catching syntax issues with your code. They analyze the code and flag errors in the syntax. JSHint output can be very illuminating. For example, the following output shows up so many things that we can change in the code. This snippet is from one of my existing projects:

```
temp git:(dev_branch) ✗ jshint test.js
test.js: line 1, col 1, Use the function form of "use strict".
test.js: line 4, col 1, 'destructuring expression' is available in
  ES6 (use esnext option) or Mozilla JS extensions (use moz).
test.js: line 44, col 70, 'arrow function syntax (=>)' is only
  available in ES6 (use esnext option).
test.js: line 61, col 33, 'arrow function syntax (=>)' is only
  available in ES6 (use esnext option).
test.js: line 200, col 29, Expected ')' to match '(' from line 200
  and instead saw ':'.
test.js: line 200, col 29, 'function closure expressions' is only
  available in Mozilla JavaScript extensions (use moz option).
test.js: line 200, col 37, Expected '}' to match '{' from line 36
  and instead saw ')'.
```

```
test.js: line 200, col 39, Expected ')' and instead saw '{'.
test.js: line 200, col 40, Missing semicolon.
```

Using strict

We briefly discussed the **strict** mode in earlier chapters. The strict mode in JavaScript flags or eliminates some of the JavaScript silent errors. Rather than silently failing, the strict mode makes these failures throw errors instead. The strict mode also helps in converting mistakes to actual errors. There are two ways of enforcing the strict mode. If you want the strict mode for the entire script, you can just add the `use strict` statement as the first line of your JavaScript program. If you want a specific function to conform with the strict mode, you can add the directive as the first line of a function:

```
function strictFn(){
// This line makes EVERYTHING under this strict mode
'use strict';
…
function nestedStrictFn() {
//Everything in this function is also nested
…
}
}
```

Runtime exceptions

These errors appear when you execute the code and try to refer to an undefined variable or process a null. When a runtime exception occurs, any code after that particular line (which caused the exception) does not get executed. It is essential to handle such exceptional scenarios correctly in the code. While exception handling can help prevent crashes, they also aid in debugging. You can wrap the code that *may* encounter a runtime exception in a `try{ }` block. When any code in this block generates a runtime exception, a corresponding handler captures it. The handler is defined by a `catch(exception){}` block. Let's clarify this using an example:

```
try {
  var a = doesnotexist; // throws a runtime exception
} catch(e) {
  console.log(e.message);  //handle the exception
  //prints - "doesnotexist is not defined"
}
```

In this example, the `var a = doesnotexist;` line tries to assign an undefined variable, `doesnotexist`, to another variable, `a`. This causes a runtime exception. When we wrap this problematic code in the `try{} catch(){}` block and when the exception occurs (or is thrown), the execution stops in the `try{}` block and goes directly to the `catch() {}` handler. The `catch` handler is responsible for handling the exceptional scenario. In this case, we are displaying the error message on the console for debugging purposes. You can explicitly throw an exception to trigger an unhandled scenario in the code. Consider the following example:

```
function engageGear(gear){
  if(gear==="R"){ console.log ("Reversing");}
  if(gear==="D"){ console.log ("Driving");}
  if(gear==="N"){ console.log ("Neutral/Parking");}
  throw new Error("Invalid Gear State");
}
try
{
  engageGear("R");  //Reversing
  engageGear("P");  //Invalid Gear State
}
catch(e){
  console.log(e.message);
}
```

In this example, we are handling valid states of a gear shift (`R`, `N`, and `D`), but when we receive an invalid state, we are explicitly throwing an exception clearly stating the reason. When we call the function that we think may throw an exception, we wrap the code in the `try{}` block and attach a `catch(){}` handler with it. When the exception is caught by the `catch()` block, we handle the exceptional condition appropriately.

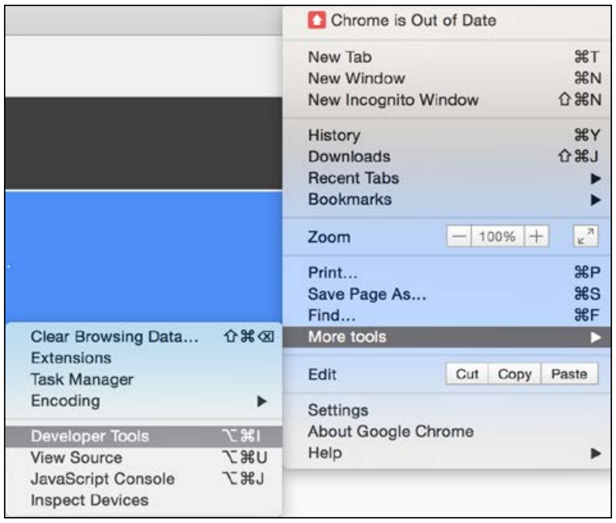
Console.log and asserts

Displaying the state of execution on the console can be very useful while debugging. However, modern developer tools allow you to put breakpoints and halt execution to inspect a particular value during runtime. You can quickly detect small issues by logging some variable state on the console.

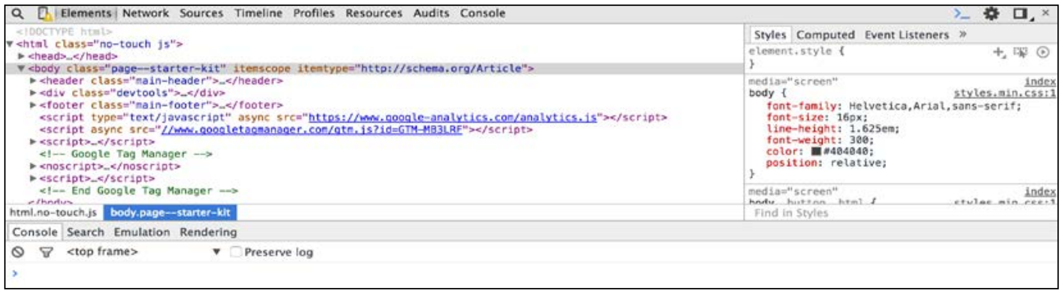
With these concepts, let's see how we can use Chrome Developer Tools to debug JavaScript code.

Chrome DevTools

You can start Chrome DevTools by navigating to menu | **More tools** | **Developer Tools**:



Chrome DevTools opens up on the lower pane of your browser and has a bunch of very useful sections:



```
<!DOCTYPE html>
▼<html class="no-touch js">
  ▶<head>…</head>
  ▼<body class="page--starter-kit" itemscope itemtype="http://schema.org/Article">
    ▶<header class="main-header">…</header>
    ▶<div class="devtools">…</div>
    ▶<footer class="main-footer">…</footer>
     <script type="text/javascript" async src="https://www.google-analytics.com/analytics.js"></script>
     <script async src="//www.googletagmanager.com/gtm.js?id=GTM-MB3LRF"></script>
    ▶<script>…</script>
     <!-- Google Tag Manager -->
    ▶<noscript>…</noscript>
    ▶<script>…</script>
     <!-- End Google Tag Manager -->
```

```
element.style {
}
media="screen"                                   index
body {                               styles.min.css:1
    font-family: Helvetica,Arial,sans-serif;
    font-size: 16px;
    line-height: 1.625em;
    font-weight: 300;
    color: ■ #404040;
    position: relative;
}
media="screen"                                   index
```

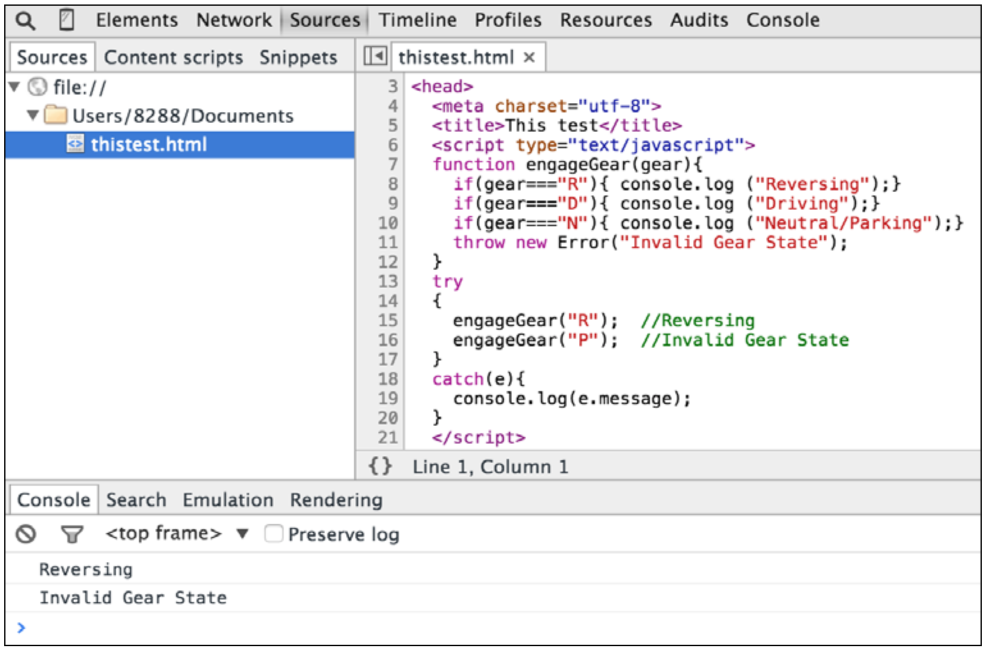
The **Elements** panel helps you inspect and monitor the DOM tree and associated style sheet for each of these components.

The **Network** panel is useful to understand network activity. For example, you can monitor the resources being downloaded over the network in real time.

The most important pane for us is the **Sources** pane. This pane is where the JavaScript source and debugger are displayed. Let's create a sample HTML with the following content:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>This test</title>
  <script type="text/javascript">
  function engageGear(gear){
    if(gear==="R"){ console.log ("Reversing");}
    if(gear==="D"){ console.log ("Driving");}
    if(gear==="N"){ console.log ("Neutral/Parking");}
    throw new Error("Invalid Gear State");
  }
  try
  {
    engageGear("R");  //Reversing
    engageGear("P");  //Invalid Gear State
  }
  catch(e){
    console.log(e.message);
  }
  </script>
</head>
<body>
</body>
</html>
```

Save this HTML file and open it in Google Chrome. Open DevTools in the browser and you will see the following screen:

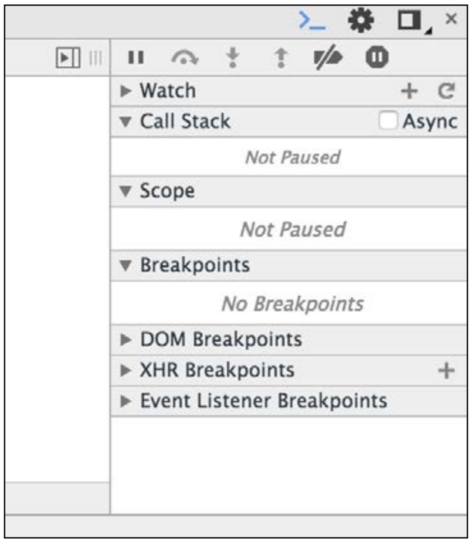


```
<head>
  <meta charset="utf-8">
  <title>This test</title>
  <script type="text/javascript">
  function engageGear(gear){
    if(gear==="R"){ console.log ("Reversing");}
    if(gear==="D"){ console.log ("Driving");}
    if(gear==="N"){ console.log ("Neutral/Parking");}
    throw new Error("Invalid Gear State");
  }
  try
  {
    engageGear("R");  //Reversing
    engageGear("P");  //Invalid Gear State
  }
  catch(e){
    console.log(e.message);
  }
  </script>
```

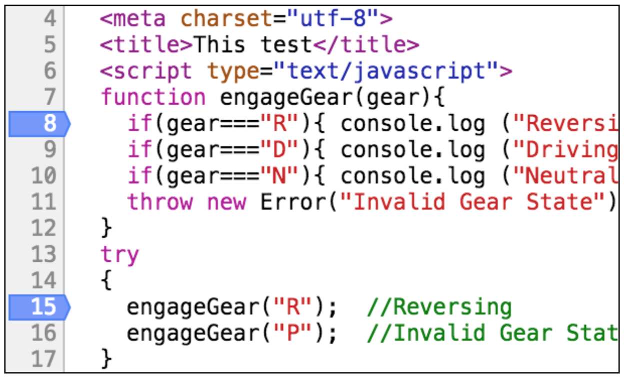
```
Reversing
Invalid Gear State
```

This is the view of the **Sources** panel. You can see the HTML and embedded JavaScript source in this panel. You can see the **Console** window as well. You can see that the file is executed and output is displayed in the **Console**.

On the right-hand side, you will see the debugger window:

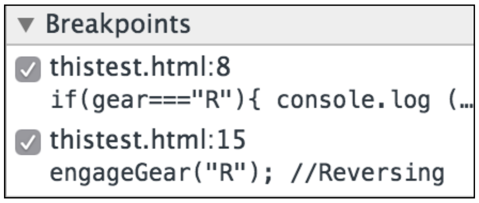


In the **Sources** panel, click on the line numbers **8** and **15** to add a breakpoint. The breakpoints allow you to stop the execution of the script at the specified point:



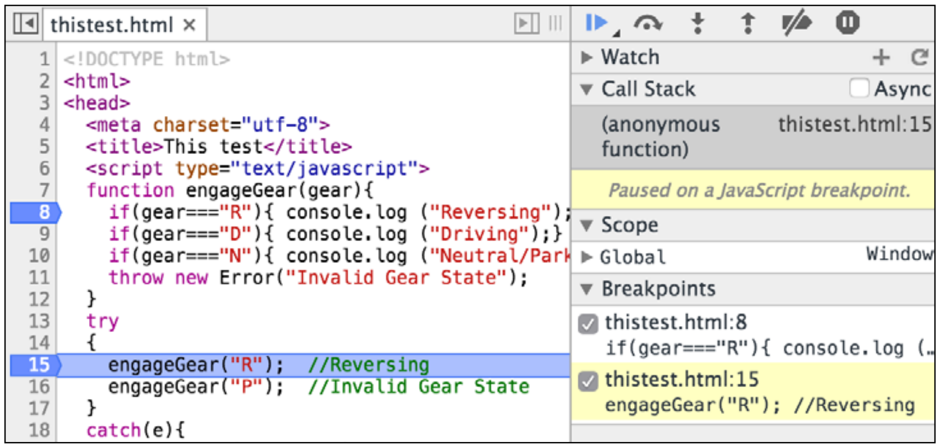
```
 4    <meta charset="utf-8">
 5    <title>This test</title>
 6    <script type="text/javascript">
 7    function engageGear(gear){
 8      if(gear==="R"){ console.log ("Reversi
 9      if(gear==="D"){ console.log ("Driving
10      if(gear==="N"){ console.log ("Neutral
11      throw new Error("Invalid Gear State")
12    }
13    try
14    {
15      engageGear("R");  //Reversing
16      engageGear("P");  //Invalid Gear Stat
17    }
```

In the debugging pane, you can see all the existing breakpoints:



```
▼ Breakpoints
☑ thistest.html:8
  if(gear==="R"){ console.log (…
☑ thistest.html:15
  engageGear("R"); //Reversing
```

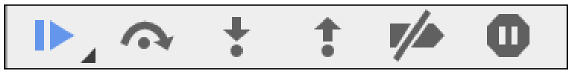
Now, when you rerun the same page, you will see that the execution stops at the debug point. One very useful technique is to inject code during the debugging phase. While the debugger is running, you can add code in order to help you understand the state of the code better:



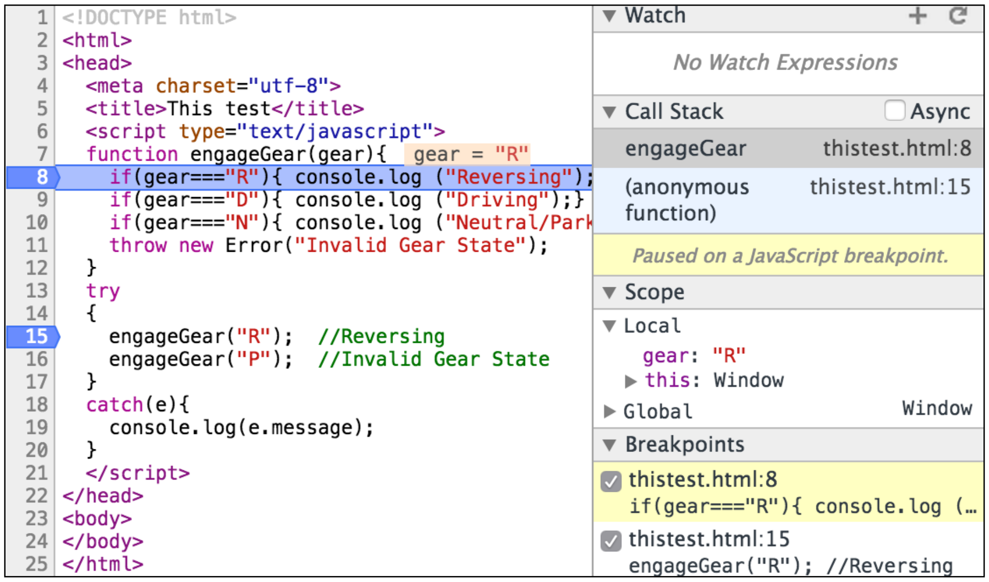
```
thistest.html ×
 1  <!DOCTYPE html>
 2  <html>
 3  <head>
 4    <meta charset="utf-8">
 5    <title>This test</title>
 6    <script type="text/javascript">
 7    function engageGear(gear){
 8      if(gear==="R"){ console.log ("Reversing");
 9      if(gear==="D"){ console.log ("Driving");}
10      if(gear==="N"){ console.log ("Neutral/Park
11      throw new Error("Invalid Gear State");
12    }
13    try
14    {
15      engageGear("R");  //Reversing
16      engageGear("P");  //Invalid Gear State
17    }
18    catch(e){
```

```
▶ Watch
▼ Call Stack                     ☐ Async
  (anonymous          thistest.html:15
  function)
  Paused on a JavaScript breakpoint.
▼ Scope
▶ Global                          Window
▼ Breakpoints
☑ thistest.html:8
  if(gear==="R"){ console.log (…
☑ thistest.html:15
  engageGear("R"); //Reversing
```

This window now has all the action. You can see that the execution is paused on line **15**. In the debug window, you can see which breakpoint is being triggered. You can see the **Call Stack** also. You can resume execution in several ways. The debug command window has a bunch of actions:



You can resume execution (which will execute until the next breakpoint) by clicking on the button. When you do this, the execution continues until the next breakpoint is encountered. In our case, we halt at line **8**:



```
 1  <!DOCTYPE html>
 2  <html>
 3  <head>
 4    <meta charset="utf-8">
 5    <title>This test</title>
 6    <script type="text/javascript">
 7    function engageGear(gear){  gear = "R"
 8      if(gear==="R"){ console.log ("Reversing");
 9      if(gear==="D"){ console.log ("Driving");}
10      if(gear==="N"){ console.log ("Neutral/Park
11      throw new Error("Invalid Gear State");
12    }
13    try
14    {
15      engageGear("R");  //Reversing
16      engageGear("P");  //Invalid Gear State
17    }
18    catch(e){
19      console.log(e.message);
20    }
21    </script>
22  </head>
23  <body>
24  </body>
25  </html>
```

```
▼ Watch
        No Watch Expressions
▼ Call Stack                     Async
  engageGear           thistest.html:8
  (anonymous           thistest.html:15
  function)
  Paused on a JavaScript breakpoint.
▼ Scope
▼ Local
    gear: "R"
  ▶ this: Window
▶ Global                         Window
▼ Breakpoints
✓ thistest.html:8
  if(gear==="R"){ console.log (…
✓ thistest.html:15
  engageGear("R"); //Reversing
```

You can observe that the **Call Stack** window shows you how we arrived at line **8**. The **Scope** panel shows the **Local** scope where you can see the variables in the scope when the breakpoint was arrived at. You can also step into or step over the next function.

There are other very useful mechanisms to debug and profile your code using Chrome DevTools. I would suggest you to go experiment with the tool and make it a part of your regular development flow.

Summary

Both the testing and debugging phases are essential to developing robust JavaScript code. TDD and BDD are approaches closely associated with the agile methodology and are widely embraced by the JavaScript developer community. In this chapter, we reviewed the best practices around TDD and usage of Jasmine as the testing framework. We saw various methods of debugging JavaScript using Chrome DevTools. In the next chapter, we will explore the new and exciting world of ES6, DOM manipulation, and cross-browser strategies.

7
ECMAScript 6

So far, we have taken a detailed tour of the JavaScript programming language. I am sure that you must have gained significant insight into the core of the language. What we saw so far was as per the **ECMAScript 5** (**ES5**) standards. **ECMAScript 6** (**ES6**) or **ECMAScript 2015** (**ES2015**) is the latest version of the ECMAScript standard. This standard is evolving and the last round of modifications was done in June, 2015. ES2015 is significant in its scope and the recommendations of ES2015 are being implemented in most JavaScript engines. This is great news. ES6 introduces a huge number of features that add syntactic forms and helpers that enrich the language significantly. The pace at which ECMAScript standards keep evolving makes it a bit difficult for browsers and JavaScript engines to support new features. It is also a practical reality that most programmers have to write code that can be supported by older browsers. The notorious Internet Explorer 6 was once the most widely used browser in the world. To make sure that your code is compatible with the most number of browsers is a daunting task. So, while you want to jump to the next set of awesome ES6 features, you will have to consider the fact that several ES6 features may not be supported by the most popular of browsers or JavaScript frameworks.

This may look like a dire scenario, but things are not that dark. **Node.js** uses the latest version of the V8 engine that supports majority of ES6 features. Facebook's **React** also supports them. Mozilla Firefox and Google Chrome are two of the most used browsers today and they support a majority of ES6 features.

To avoid such pitfalls and unpredictability, certain solutions have been proposed. The most useful among these are polyfills/shims and transpilers.

Shims or polyfills

Polyfills (also known as shims) are patterns to define behavior from a new version in a compatible form supported by an older version of the environment. There's a great collection of ES6 shims called **ES6 shim** (`https://github.com/paulmillr/es6-shim/`); I would highly recommend a study of these shims. From the ES6 shim collection, consider the following example of a shim.

The `Number.isFinite()` method of the ECMAScript 2015 (ES6) standard determines whether the passed value is a finite number. The equivalent shim for it would look something as follows:

```
var numberIsFinite = Number.isFinite || function isFinite(value) {
  return typeof value === 'number' && globalIsFinite(value);
};
```

The shim first checks if the `Number.isFinite()` method is available; if not, it *fills* it up with an implementation. This is a pretty nifty technique to fill in gaps in specifications. Shims are constantly upgraded with newer features and, hence, it is a sound strategy to keep the most updated shims in your project.

> The `endsWith()` polyfill is described in detail at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/endsWith`. `String.endsWith()` is part of ES6 but can be polyfilled easily for pre-ES6 environments.

Shims, however, cannot polyfill syntactical changes. For this, we can consider transpilers as an option.

Transpilers

Transpiling is a technique that combines both compilation and transformation. The idea is to write ES6-compatible code and use a tool that transpiles this code into a valid and equivalent ES5 code. We will be looking at the most complete and popular transpiler for ES6 called **Babel** (`https://babeljs.io/`).

Babel can be used in various ways. You can install it as a node module and invoke it from the command line or import it as a script in your web page. Babel's setup is exhaustive and well-documented at `https://babeljs.io/docs/setup/`. Babel also has a great **Read-Eval-Print-Loop** (**REPL**). We will Babel REPL for most of the examples in this chapter. An in-depth understanding of various ways in which Babel can be used is out of the scope of this book. However, I would urge you to start using Babel as part of your development workflow.

We will cover the most important part of ES6 specifications in this chapter. You should explore all the features of ES6 if possible and make them part of your development workflow.

ES6 syntax changes

ES6 brings in significant syntactic changes to JavaScript. These changes need careful study and some getting used to. In this section, we will study some of the most important syntax changes and see how you can use Babel to start using these newer constructs in your code right away.

Block scoping

We discussed earlier that the variables in JavaScript are function-scoped. Variables created in a nested scope are available to the entire function. Several programming languages provide you with a default block scope where any variable declared within a block of code (usually delimited by `{}`) is scoped (available) only within this block. To achieve a similar block scope in JavaScript, a prevalent method is to use **immediately-invoked function expressions** (**IIFE**). Consider the following example:

```
var a = 1;
(function blockscope(){
    var a = 2;
    console.log(a);   // 2
})();
console.log(a);       // 1
```

Using the IIFE, we are creating a block scope for the `a` variable. When a variable is declared in the IIFE, its scope is restricted within the function. This is the traditional way of simulating the block scope. ES6 supports block scoping without using IIFEs. In ES6, you can enclose any statement(s) in a block defined by `{}`. Instead of using `var`, you can declare a variable using `let` to define the block scope. The preceding example can be rewritten using ES6 block scopes as follows:

```
"use strict";
var a = 1;
{
  let a = 2;
  console.log( a ); // 2
}
console.log( a ); // 1
```

Using standalone brackets `{}` may seem unusual in JavaScript, but this convention is fairly common to create a block scope in many languages. The block scope kicks in other constructs such as `if { }` or `for (){ }` as well.

When you use a block scope in this way, it is generally preferred to put the variable declaration on top of the block. One difference between variables declared using `var` and `let` is that variables declared with `var` are attached to the entire function scope, while variables declared using `let` are attached to the block scope and they are not initialized until they appear in the block. Hence, you cannot access a variable declared with `let` earlier than its declaration, whereas with variables declared using `var`, the ordering doesn't matter:

```
function fooey() {
  console.log(foo); // ReferenceError
  let foo = 5000;
}
```

One specific use of `let` is in for loops. When we use a variable declared using `var` in a for loop, it is created in the global or parent scope. We can create a block-scoped variable in the for loop scope by declaring a variable using `let`. Consider the following example:

```
for (let i = 0; i<5; i++) {
  console.log(i);
}
console.log(i); // i is not defined
```

As `i` is created using `let`, it is scoped in the `for` loop. You can see that the variable is not available outside the scope.

One more use of block scopes in ES6 is the ability to create constants. Using the `const` keyword, you can create constants in the block scope. Once the value is set, you cannot change the value of such a constant:

```
if(true){
  const a=1;
  console.log(a);
  a=100;  ///"a" is read-only, you will get a TypeError
}
```

A constant has to be initialized while being declared. The same block scope rules apply to functions also. When a function is declared inside a block, it is available only within that scope.

Default parameters

Defaulting is very common. You always set some default value to parameters passed to a function or variables that you initialize. You may have seen code similar to the following:

```
function sum(a,b){
  a = a || 0;
  b = b || 0;
  return (a+b);
}
console.log(sum(9,9)); //18
console.log(sum(9));   //9
```

Here, we are using `||` (the OR operator) to default variables `a` and `b` to `0` if no value was supplied when this function was invoked. With ES6, you have a standard way of defaulting function arguments. The preceding example can be rewritten as follows:

```
function sum(a=0, b=0){
  return (a+b);
}
console.log(sum(9,9)); //18
console.log(sum(9));   //9
```

You can pass any valid expression or function call as part of the default parameter list.

Spread and rest

ES6 has a new operator, ... Based on how it is used, it is called either `spread` or `rest`. Let's look at a trivial example:

```
function print(a, b){
  console.log(a,b);
}
print(...[1,2]);  //1,2
```

What's happening here is that when you add ... before an array (or an iterable) it *spreads* the element of the array in individual variables in the function parameters. The `a` and `b` function parameters were assigned two values from the array when it was spread out. Extra parameters are ignored while spreading an array:

```
print(...[1,2,3 ]);  //1,2
```

This would still print `1` and `2` because there are only two functional parameters available. Spreads can be used in other places also, such as array assignments:

```
var a = [1,2];
var b = [ 0, ...a, 3 ];
console.log( b ); //[0,1,2,3]
```

There is another use of the ... operator that is the very opposite of the one that we just saw. Instead of spreading the values, the same operator can gather them into one:

```
function print (a,...b){
  console.log(a,b);
}
console.log(print(1,2,3,4,5,6,7));  //1 [2,3,4,5,6,7]
```

In this case, the variable `b` takes the *rest* of the values. The `a` variable took the first value as `1` and `b` took the rest of the values as an array.

Destructuring

If you have worked on a functional language such as **Erlang**, you will relate to the concept of pattern matching. Destructuring in JavaScript is something very similar. Destructuring allows you to bind values to variables using pattern matching. Consider the following example:

```
var [start, end] = [0,5];
for (let i=start; i<end; i++){
  console.log(i);
}
//prints - 0,1,2,3,4
```

We are assigning two variables with the help of array destructuring:

```
var [start, end] = [0,5];
```

As shown in the preceding example, we want the pattern to match when the first value is assigned to the first variable (`start`) and the second value is assigned to the second variable (`end`). Consider the following snippet to see how the destructuring of array elements works:

```
function fn() {
  return [1,2,3];
}
var [a,b,c]=fn();
console.log(a,b,c); //1 2 3
```

```
//We can skip one of them
var [d,,f]=fn();
console.log(d,f);   //1 3
//Rest of the values are not used
var [e,] = fn();
console.log(e);     //1
```

Let's discuss how objects' destructuring works. Let's say that you have a function `f` that returns an object as follows:

```
function f() {
  return {
    a: 'a',
    b: 'b',
    c: 'c'
  };
}
```

When we destructure the object being returned by this function, we can use the similar syntax as we saw earlier; the difference is that we use {} instead of `[]`:

```
var { a: a, b: b, c: c } = f();
console.log(a,b,c); //a b c
```

Similar to arrays, we use pattern matching to assign variables to their corresponding values returned by the function. There is an even shorter way of writing this if you are using the same variable as the one being matched. The following example would do just fine:

```
var { a,b,c } = f();
```

However, you would mostly be using a different variable name from the one being returned by the function. It is important to remember that the syntax is *source: destination* and not the usual *destination: source*. Carefully observe the following example:

```
//this is target: source - which is incorrect
var { x: a, x: b, x: c } = f();
console.log(x,y,z); //x is undefined, y is undefined z is undefined
//this is source: target - correct
var { a: x, b: y, c: z } = f();
console.log(x,y,z); // a b c
```

This is the opposite of the *target* = *source* way of assigning values and hence will take some time in getting used to.

Object literals

Object literals are everywhere in JavaScript. You would think that there is no scope of improvement there. However, ES6 wants to improve this too. ES6 introduces several shortcuts to create a concise syntax around object literals:

```
var firstname = "Albert", lastname = "Einstein",
  person = {
    firstname: firstname,
    lastname: lastname
  };
```

If you intend to use the same property name as the variable that you are assigning, you can use the concise property notation of ES6:

```
var firstname = "Albert", lastname = "Einstein",
  person = {
    firstname,
    lastname
  };
```

Similarly, you are assigning functions to properties as follows:

```
var person = {
  getName: function(){
    // ..
  },
  getAge: function(){
    //..
  }
}
```

Instead of the preceding lines, you can say the following:

```
var person = {
  getName(){
    // ..
  },
  getAge(){
    //..
  }
}
```

Template literals

I am sure you have done things such as the following:

```
function SuperLogger(level, clazz, msg){
  console.log(level+": Exception happened in class:"+clazz+" -
    Exception :"+ msg);
}
```

This is a very common way of replacing variable values to form a string literal. ES6 provides you with a new type of string literal using the backtick (`) delimiter. You can use string interpolation to put placeholders in a template string literal. The placeholders will be parsed and evaluated.

The preceding example can be rewritten as follows:

```
function SuperLogger(level, clazz, msg){
  console.log(`${level} : Exception happened in class: ${clazz} -
    Exception : {$msg}`);
}
```

We are using `` ` ` `` around a string literal. Within this literal, any expression of the `${..}` form is parsed immediately. This parsing is called interpolation. While parsing, the variable's value replaces the placeholder within `${}`. The resulting string is just a normal string with the placeholders replaced with actual variable values.

With string interpolation, you can split a string into multiple lines also, as shown in the following code (very similar to Python):

```
var quote =
`Good night, good night!
Parting is such sweet sorrow,
that I shall say good night
till it be morrow.`;
console.log( quote );
```

You can use function calls or valid JavaScript expressions as part of the string interpolation:

```
function sum(a,b){
  console.log(`The sum seems to be ${a + b}`);
}
sum(1,2); //The sum seems to be 3
```

The final variation of the template strings is called **tagged template string**. The idea is to modify the template string using a function. Consider the following example:

```
function emmy(key, ...values){
  console.log(key);
  console.log(values);
}
let category="Best Movie";
let movie="Adventures in ES6";
emmy`And the award for ${category} goes to ${movie}`;

//["And the award for "," goes to ",""]
//["Best Movie","Adventures in ES6"]
```

The strangest part is when we call the `emmy` function with the template literal. It's not a traditional function call syntax. We are not writing `emmy();` we are just *tagging* the literal with the function. When this function is called, the first argument is an array of all the plain strings (the string between interpolated expressions). The second argument is the array where all the interpolated expressions are evaluated and stored.

Now what this means is that the tag function can actually change the resulting template tag:

```
function priceFilter(s, ...v){
  //Bump up discount
  return s[0]+ (v[0] + 5);
}
let default_discount = 20;
let greeting = priceFilter `Your purchase has a discount of
  ${default_discount} percent`;
console.log(greeting);  //Your purchase has a discount of 25
```

As you can see, we modified the value of the discount in the tag function and returned the modified values.

Maps and Sets

ES6 introduces four new data structures: **Map**, **WeakMap**, **Set**, and **WeakSet**. We discussed earlier that objects are the usual way of creating key-value pairs in JavaScript. The disadvantage of objects is that you cannot use non-string values as keys. The following snippets demonstrate how Maps are created in ES6:

```
let m = new Map();
let s = { 'seq' : 101 };

m.set('1','Albert');
m.set('MAX', 99);
m.set(s,'Einstein');

console.log(m.has('1')); //true
console.log(m.get(s));   //Einstein
console.log(m.size);     //3
m.delete(s);
m.clear();
```

You can initialize the map while declaring it:

```
let m = new Map([
  [ 1, 'Albert' ],
  [ 2, 'Douglas' ],
  [ 3, 'Clive' ],
]);
```

If you want to iterate over the entries in the Map, you can use the `entries()` function that will return you an iterator. You can iterate through all the keys using the `keys()` function and you can iterate through the values of the Map using the `values()` function:

```
let m2 = new Map([
    [ 1, 'Albert' ],
    [ 2, 'Douglas' ],
    [ 3, 'Clive' ],
]);
for (let a of m2.entries()){
  console.log(a);
}
//[1,"Albert"] [2,"Douglas"][3,"Clive"]
for (let a of m2.keys()){
  console.log(a);
} //1 2 3
for (let a of m2.values()){
  console.log(a);
}
//Albert Douglas Clive
```

A variation of JavaScript Maps is a WeakMap—a WeakMap does not prevent its keys from being garbage-collected. Keys for a WeakMap must be objects and the values can be arbitrary values. While a WeakMap behaves in the same way as a normal Map, you cannot iterate through it and you can't clear it. There are reasons behind these restrictions. As the state of the Map is not guaranteed to remain static (keys may get garbage-collected), you cannot ensure correct iteration.

There are not many cases where you may want to use WeakMap. Most uses of a Map can be written using normal Maps.

While Maps allow you to store arbitrary values, Sets are a collection of unique values. Sets have similar methods as Maps; however, `set()` is replaced with `add()`, and the `get()` method does not exist. The reason that the `get()` method is not there is because a Set has unique values, so you are interested in only checking whether the Set contains a value or not. Consider the following example:

```
let x = {'first': 'Albert'};
let s = new Set([1,2,'Sunday',x]);
//console.log(s.has(x));  //true
s.add(300);
//console.log(s);  //[1,2,"Sunday",{"first":"Albert"},300]

for (let a of s.entries()){
  console.log(a);
}
//[1,1]
//[2,2]
//["Sunday","Sunday"]
//[{"first":"Albert"},{"first":"Albert"}]
//[300,300]
for (let a of s.keys()){
  console.log(a);
}
//1
//2
//Sunday
//{"first":"Albert"}
//300
for (let a of s.values()){
  console.log(a);
}
//1
//2
//Sunday
//{"first":"Albert"}
//300
```

The `keys()` and `values()` iterators both return a list of the unique values in the Set. The `entries()` iterator yields a list of entry arrays, where both items of the array are the unique Set values. The default iterator for a Set is its `values()` iterator.

Symbols

ES6 introduces a new data type called Symbol. A Symbol is guaranteed to be unique and immutable. Symbols are usually used as an identifier for object properties. They can be considered as uniquely generated IDs. You can create Symbols with the `Symbol()` factory method—remember that this is not a constructor and hence you should not use a `new` operator:

```
let s = Symbol();
console.log(typeof s); //symbol
```

Unlike strings, Symbols are guaranteed to be unique and hence help in preventing name clashes. With Symbols, we have an extensibility mechanism that works for everyone. ES6 comes with a number of predefined built-in Symbols that expose various meta behaviors on JavaScript object values.

Iterators

Iterators have been around in other programming languages for quite some time. They give convenience methods to work with collections of data. ES6 introduces iterators for the same use case. ES6 iterators are objects with a specific interface. Iterators have a `next()` method that returns an object. The returning object has two properties—`value` (the next value) and `done` (indicates whether the last result has been reached). ES6 also defines an `Iterable` interface, which describes objects that must be able to produce iterators. Let's look at an array, which is an iterable, and the iterator that it can produce to consume its values:

```
var a = [1,2];
var i = a[Symbol.iterator]();
console.log(i.next());      // { value: 1, done: false }
console.log(i.next());      // { value: 2, done: false }
console.log(i.next());      // { value: undefined, done: true }
```

As you can see, we are accessing the array's iterator via `Symbol.iterator()` and calling the `next()` method on it to get each successive element. Both `value` and `done` are returned by the `next()` method call. When you call `next()` past the last element in the array, you get an undefined value and `done: true`, indicating that you have iterated over the entire array.

For..of loops

ES6 adds a new iteration mechanism in form of the `for..of` loop, which loops over the set of values produced by an iterator.

The value that we iterate over with `for..of` is an iterable.

Let's compare `for..of` to `for..in`:

```
var list = ['Sunday','Monday','Tuesday'];
for (let i in list){
  console.log(i);  //0 1 2
}
for (let i of list){
  console.log(i);  //Sunday Monday Tuesday
}
```

As you can see, using the `for..in` loop, you can iterate over indexes of the `list` array, while the `for..of` loop lets you iterate over the values stored in the `list` array.

Arrow functions

One of the most interesting new parts of ECMAScript 6 is arrow functions. Arrow functions are, as the name suggests, functions defined with a new syntax that uses an *arrow* (`=>`) as part of the syntax. Let's first see how arrow functions look:

```
//Traditional Function
function multiply(a,b) {
  return a*b;
}
//Arrow
var multiply = (a,b) => a*b;
console.log(multiply(1,2)); //2
```

The arrow function definition consists of a parameter list (of zero or more parameters and surrounding `( .. )` if there's not exactly one parameter), followed by the `=>` marker, which is followed by a function body.

The body of the function can be enclosed by `{ .. }` if there's more than one expression in the body. If there's only one expression, and you omit the surrounding `{ .. }`, there's an implied return in front of the expression. There are several variations of how you can write arrow functions. The following are the most commonly used:

```
// single argument, single statement
//arg => expression;
var f1 = x => console.log("Just X");
f1(); //Just X

// multiple arguments, single statement
//(arg1 [, arg2]) => expression;
var f2 = (x,y) => x*y;
console.log(f2(2,2)); //4

// single argument, multiple statements
// arg => {
//     statements;
// }
var f3 = x => {
  if(x>5){
    console.log(x);
  }
  else {
    console.log(x+5);
  }
}
f3(6); //6

// multiple arguments, multiple statements
// ([arg] [, arg]) => {
//   statements
// }
var f4 = (x,y) => {
  if(x!=0 && y!=0){
    return x*y;
  }
}
console.log(f4(2,2));//4
```

```
// with no arguments, single statement
//() => expression;
var f5 = () => 2*2;
console.log(f5()); //4

//IIFE
console.log(( x => x * 3 )( 3 )); // 9
```

It is important to remember that all the characteristics of a normal function parameter are available to arrow functions, including default values, destructuring, and rest parameters.

Arrow functions offer a convenient and short syntax, which gives your code a very *functional programming* flavor. Arrow functions are popular because they offer an attractive promise of writing concise functions by dropping function, return, and { .. } from the code. However, arrow functions are designed to fundamentally solve a particular and common pain point with this-aware coding. In normal ES5 functions, every new function defined its own value of `this` (a new object in case of a constructor, `undefined` in strict mode function calls, context object if the function is called as an *object method*, and so on). JavaScript functions always have their own `this` and this prevents you from accessing the `this` of, for example, a surrounding method from inside a callback. To understand this problem, consider the following example:

```
function CustomStr(str){
  this.str = str;
}
CustomStr.prototype.add = function(s){   // --> 1
  'use strict';
  return s.map(function (a){             // --> 2
    return this.str + a;                 // --> 3
  });
};

var customStr = new CustomStr("Hello");
console.log(customStr.add(["World"]));
//Cannot read property 'str' of undefined
```

On the line marked with `3`, we are trying to get `this.str`, but the anonymous function also has its own `this`, which shadows `this` from the method from line `1`. To fix this in ES5, we can assign `this` to a variable and use the variable instead:

```
function CustomStr(str){
  this.str = str;
}
```

```
CustomStr.prototype.add = function(s){
  'use strict';
  var that = this;                        // --> 1
  return s.map(function (a){              // --> 2
    return that.str + a;                  // --> 3
  });
};

var customStr = new CustomStr("Hello");
console.log(customStr.add(["World"]));
//["HelloWorld]
```

On the line marked with `1`, we are assigning `this` to a variable, `that`, and in the anonymous function we are using the `that` variable, which will have a reference to `this` from the correct context.

ES6 arrow functions have lexical `this`, meaning that the arrow functions capture the `this` value of the enclosing context. We can convert the preceding function to an equivalent arrow function as follows:

```
function CustomStr(str){
  this.str = str;
}
CustomStr.prototype.add = function(s){
  return s.map((a)=> {
    return this.str + a;
  });
};
var customStr = new CustomStr("Hello");
console.log(customStr.add(["World"]));
//["HelloWorld]
```

Summary

In this chapter, we discussed a few important features being added to the language in ES6. It's an exciting collection of new language features and paradigms and, using polyfills and transpilers, you can start with them right away. JavaScript is an ever growing language and it is important to understand what the future holds. ES6 features make JavaScript an even more interesting and mature language. In the next chapter, we will dive deep into manipulating the browser's **Document Object Model** (**DOM**) and events using JavaScript with jQuery.

8 DOM Manipulation and Events

The most important reason for JavaScript's existence is the web. JavaScript is the language for the web and the browser is the raison d'être for JavaScript. JavaScript gives dynamism to otherwise static web pages. In this chapter, we will dive deep into this relationship between the browser and language. We will understand the way in which JavaScript interacts with the components of the web page. We will look at the **Document Object Model** (**DOM**) and JavaScript event model.

DOM

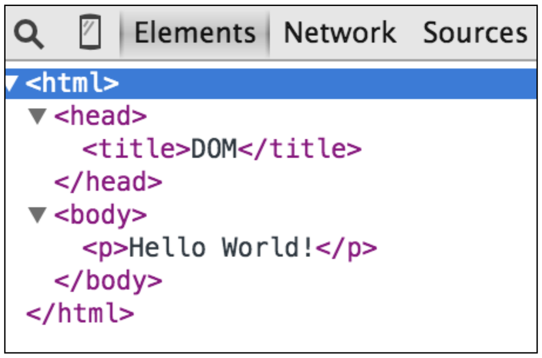
In this chapter, we will look at various aspects of JavaScript with regard to the browser and HTML. HTML, as I am sure you are aware, is the markup language used to define web pages. Various forms of markups exist for different uses. The popular marks are **Extensible Markup Language** (**XML**) and **Standard Generalized Markup Language** (**SGML**). Apart from these generic markup languages, there are very specific markup languages for specific purposes such as text processing and image meta information. **HyperText Markup Language** (**HTML**) is the standard markup language that defines the presentation semantics of a web page. A web page is essentially a document. The DOM provides you with a representation of this document. The DOM also provides you with a means of storing and manipulating this document. The DOM is the programming interface of HTML and allows structural manipulation using scripting languages such as JavaScript. The DOM provides a structural representation of the document. The structure consists of nodes and objects. Nodes have properties and methods on which you can operate in order to manipulate the nodes themselves. The DOM is just a representation and not a programming construct. DOM acts as a model for DOM processing languages such as JavaScript.

Accessing DOM elements

Most of the time, you will be interested in accessing DOM elements to inspect their values or processing these values for some business logic. We will take a detailed look at this particular use case. Let's create a sample HTML file with the following content:

```
<html>
<head>
  <title>DOM</title>
</head>
<body>
  <p>Hello World!</p>
</body>
</html>
```

You can save this file as `sample_dom.html`; when you open this in the Google Chrome browser, you will see the web page displayed with the **Hello World** text displayed. Now, open Google Chrome Developer Tools by navigating to options | **More Tools** | **Developer Tools** (this route may differ on your operating system and browser version). In the **Developer Tools** window, you will see the DOM structure:



```
▼<html>
  ▼<head>
     <title>DOM</title>
   </head>
  ▼<body>
     <p>Hello World!</p>
   </body>
 </html>
```

Next, we will insert some JavaScript into this HTML page. We will invoke the JavaScript function when the web page is loaded. To do this, we will call a function on `window.onload`. You can place your script in the `<script>` tag located under the `<head>` tag. Your page should look as follows:

```
<html>
  <head>
    <title>DOM</title>
    <script>
```

```
        // run this function when the document is loaded
        window.onload = function() {
          var doc = document.documentElement;
          var body = doc.body;
          var _head = doc.firstChild;
          var _body = doc.lastChild;
          var _head_ = doc.childNodes[0];
          var title = _head.firstChild;
          alert(_head.parentNode === doc); //true
        }
      </script>
    </head>
    <body>
      <p>Hello World!</p>
    </body>
  </html>
```

The anonymous function is executed when the browser loads the page. In the function, we are getting the nodes of the DOM programmatically. The entire HTML document can be accessed using the `document.documentElement` function. We store the document in a variable. Once the document is accessed, we can traverse the nodes using several helper properties of the document. We are accessing the `<body>` element using `doc.body`. You can traverse through the children of an element using the `childNodes` array. The first and last children of a node can be accessed using additional properties—`firstChild` and `lastChild`.

> It is not recommended to use render-blocking JavaScript in the `<head>` tag. This slows down the page render dramatically. Modern browsers support the `async` and `defer` attributes to indicate to the browsers that the rendering can go on while the script is being downloaded. You can use these tags in the `<head>` tag without worrying about performance degradation. You can get more information at `http://stackoverflow.com/questions/436411/where-is-the-best-place-to-put-script-tags-in-html-markup`.

Accessing specific nodes

The core DOM defines the `getElementsByTagName()` method to return `NodeList` of all the element objects whose `tagName` property is equal to a specific value. The following line of code returns a list of all the `<p/>` elements in a document:

```
  var paragraphs = document.getElementsByTagName('p');
```

The HTML DOM defines `getElementsByName()` to retrieve all the elements that have their name attribute set to a specific value. Consider the following snippet:

```
<html>
  <head>
    <title>DOM</title>
    <script>
      showFeelings = function() {
        var feelings = document.getElementsByName("feeling");
        alert(feelings[0].getAttribute("value"));
        alert(feelings[1].getAttribute("value"));
      }
    </script>
  </head>
  <body>
    <p>Hello World!</p>
    <form method="post" action="/post">
      <fieldset>
        <p>How are you feeling today?</p>
        <input type="radio" name="feeling" value="Happy" />
          Happy<br />
        <input type="radio" name="feeling" value="Sad" />Sad<br />
      </fieldset>
      <input type="button" value="Submit"
        onClick="showFeelings()"/>
    </form>
  </body>
</html>
```

In this example, we are creating a group of radio buttons with the `name` attribute defined as `feeling`. In the `showFeelings` function, we get all the elements with the `name` attribute set to `feeling` and we iterate through all these elements.

The other method defined by the HTML DOM is `getElementById()`. This is a very useful method in accessing a specific element. This method does the lookup based on the `id` associated with an element. The `id` attribute is unique for every element and, hence, this kind of lookup is very fast and should be preferred over `getElementsByName()`. -However, you should be aware that the browser does not guarantee the uniqueness of the `id` attribute. In the following example, we are accessing a specific element using the ID. Element IDs are unique as opposed to tags or name attributes:

```
<html>
  <head>
```

```
      <title>DOM</title>
      <script>
        window.onload= function() {
          var greeting = document.getElementById("greeting");
          alert(greeting.innerHTML); //shows "Hello World" alert
        }
      </script>
    </head>
    <body>
      <p id="greeting">Hello World!</p>
      <p id="identify">Earthlings</p>
    </body>
  </html>
```

What we discussed so far was the basics of DOM traversal in JavaScript. When the DOM gets complex and you want sophisticated operations on the DOM, these traversal and access functions seem limiting. With this basic knowledge with us, it's time to get introduced to a fantastic library for DOM traversal (among other things) called jQuery.

jQuery is a lightweight library designed to make common browser operations easier. Common operations such as DOM traversal and manipulation, event handling, animation, and Ajax can be tedious if done using pure JavaScript. jQuery provides you with easy-to-use and shorter helper mechanisms to help you develop these common operations very easily and quickly. jQuery is a feature-rich library, but as far as this chapter goes, we will focus primarily on DOM manipulation and events.

You can add jQuery to your HTML by adding the script directly from a **content delivery network** (**CDN**) or manually downloading the file and adding it to the script tag. The following example shows you how to download jQuery from Google's CDN:

```
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/
      jquery/2.1.4/jquery.min.js"></script>
  </head>
  <body>
  </body>
</html>
```

The advantage of a CDN download is that Google's CDN automatically finds the nearest download server for you and keeps an updated stable copy of the jQuery library. If you wish to download and manually host jQuery along with your website, you can add the script as follows:

```
<script src="./lib/jquery.js"></script>
```

In this example, the jQuery library is manually downloaded in the `lib` directory. With the jQuery setup in the HTML page, let's explore the methods of manipulating the DOM elements. Consider the following example:

```
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/
      jquery/2.1.4/jquery.min.js"></script>
    <script>
    $(document).ready(function() {
        $('#greeting').html('Hello World Martian');
    });
  </script>
  </head>
  <body>
    <p id="greeting">Hello World Earthling ! </p>
  </body>
</html>
```

After adding jQuery to the HTML page, we write the custom JavaScript that selects the element with a `greeting` ID and changes its value. The strange-looking code within `$()` is the jQuery in action. If you read the jQuery source code (and you should, it's brilliant) you will see the final line:

```
// Expose jQuery to the global object
window.jQuery = window.$ = jQuery;
```

The `$` is just a function. It is an alias for the function called jQuery. The `$` is a syntactic sugar that makes the code concise. In fact, you can use both `$` and `jQuery` interchangeably. For example, both `$('#greeting').html('Hello World Martian');` and `jQuery('#greeting').html('Hello World Martian');` are the same.

You can't use jQuery before the page is completely loaded. As jQuery will need to know all the nodes of the DOM structure, the entire DOM has to be in-memory. To ensure that the page is completely loaded and in a state where it's ready to be manipulated, we can use the `$(document).ready()` function. Here, the IIFE is executed only after the entire documented is *ready*:

```
$(document).ready(function() {
  $('#greeting').html('Hello World Martian');
});
```

This snippet shows you how we can associate a function to jQuery's `.ready()` function. This function will be executed once the document is ready. We are using `$(document)` to create a jQuery object from our page's document. We are calling the `.ready()` function on the jQuery object and passing it the function that we want to execute.

This is a very common thing to do when using jQuery—so much so that it has its own shortcut. You can replace the entire `ready()` call with a short `$()` call:

```
$(function() {
  $('#greeting').html('Hello World Martian');
});
```

The most important function in jQuery is `$()`. This function typically accepts a CSS selector as its sole parameter and returns a new jQuery object pointing to the corresponding elements on the page. The three primary selectors are the tag name, ID, and class. They can be used either on their own or in combination with others. The following simple examples illustrate how these three selectors appear in code:

Selector	**CSS Selector**	**jQuery Selector**	**Output from the selector**
Tag	`p{}`	`$('p')`	This selects all the `p` tags from the document.
Id	`#div_1`	`$('#div_1')`	This selects single elements that have a `div_1` ID. The symbol used to identify the ID is `#`.
Class	`.bold_fonts`	`$('.bold_fonts')`	This selects all the elements in the document that have the CSS class `bold_fonts`. The symbol used to identify the class match is "`.`".

jQuery works on CSS selectors.

> As CSS selectors are not in the scope of this book, I would suggest that you go to `http://www.w3.org/TR/CSS2/selector.html` to get a fair idea of the concept.

We also assume that you are familiar with HTML tags and syntax. The following example covers the fundamental idea of how jQuery selectors work:

```
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/
jquery.min.js"></script>
    <script>
      $(function() {
        $('h1').html(function(index, oldHTML){
          return oldHTML + "Finally?";
        });
        $('h1').addClass('highlight-blue');
        $('#header > h1 ').css('background-color', 'cyan');
        $('ul li:not(.highlight-blue)').addClass(
          'highlight-green');
        $('tr:nth-child(odd)').addClass('zebra');
      });
    </script>
    <style>
      .highlight-blue {
        color: blue;
      }
      .highlight-green{
        color: green;
      }
      .zebra{
        background-color: #666666;
        color: white;
      }
    </style>
  </head>
  <body>
```

```
        <div id=header>
          <h1>Are we there yet ? </h1>
          <span class="highlight">
            <p>Journey to Mars</p>
            <ul>
              <li>First</li>
              <li>Second</li>
              <li class="highlight-blue">Third</li>
            </ul>
          </span>
          <table>
            <tr><th>Id</th><th>First name</th><th>Last Name</th></tr>
            <tr><td>1</td><td>Albert</td><td>Einstein</td></tr>
            <tr><td>2</td><td>Issac</td><td>Newton</td></tr>
            <tr><td>3</td><td>Enrico</td><td>Fermi</td></tr>
            <tr><td>4</td><td>Richard</td><td>Feynman</td></tr>
          </table>
        </div>
      </body>
    </html>
```

In this example, we are selecting several DOM elements in the HTML page using selectors. We have an H1 header with the text, `Are we there yet ?`; when the page loads, our jQuery script accesses all H1 headers and appends the text `Finally?` to them:

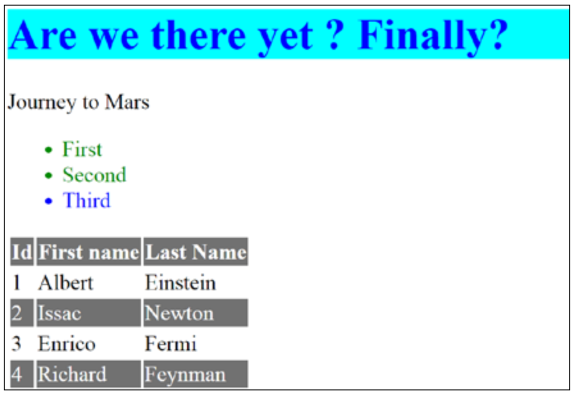
```
$('h1').html(function(index, oldHTML){
  return oldHTML + "Finally ?";
});
```

The `$.html()` function sets the HTML for the target element—an H1 header in this case. Additionally, we select all H1 headers and apply a specific CSS style class, `highlight-blue`, to all of them. The `$('h1').addClass('highlight-blue')` statement selects all the H1 headers and uses the `$.addClass(<CSS class>)` method to apply a CSS class to all the elements selected using the selector.

We use the child combinator (`>`) to custom CSS styles using the `$.css()` function. In effect, the selector in the `$()` function is saying, "Find each header (`h1`) that is a child (`>`) of the element with an ID of header (`#header`)." For each such element, we apply a custom CSS. The next usage is interesting. Consider the following line:

```
$('ul li:not(.highlight-blue)').addClass('highlight-green');
```

We are selecting "For all list elements (`li`) that do not have the class `highlight-blue` applied to them, apply CSS class `highlight-green`. The final line—`$('tr:nth-child(odd)').addClass('zebra')`—can be interpreted as: From all table rows (`tr`), for every odd row, apply CSS style `zebra`. The *n*th-child selector is a custom selector provided by jQuery. The final output looks something similar to the following (Though it shows several jQuery selector types, it is very clear that knowledge of jQuery is not a substitute for bad design taste.):



Are we there yet ? Finally?

Journey to Mars

- First
- Second
- Third

Id	First name	Last Name
1	Albert	Einstein
2	Issac	Newton
3	Enrico	Fermi
4	Richard	Feynman

Once you have made a selection, there are two broad categories of methods that you can call on the selected element. These methods are **getters** and **setters**. Getters retrieve a piece of information from the selection, and setters alter the selection in some way.

Getters usually operate only on the first element in a selection while setters operate on all the elements in a selection. Setters use implicit iteration to automatically iterate over all the elements in the selection.

For example, we want to apply a CSS class to all list items on the page. When we call the `addClass` method on the selector, it is automatically applied to all elements of this particular selection. This is implicit iteration in action:

```
$( 'li' ).addClass( highlighted' );
```

However, sometimes you just don't want to go through all the elements via implicit iteration. You may want to selectively modify only a few of the elements. You can explicitly iterate over the elements using the `.each()` method. In the following code, we are processing elements selectively and using the `index` property of the element:

```
$( 'li' ).each(function( index, element ) {
  if(index % 2 == 0)
    $(elem).prepend( '<b>' + STATUS + '</b>' );
});
```

Chaining

Chaining jQuery methods allows you to call a series of methods on a selection without temporarily storing the intermediate values. This is possible because every setter method that we call returns the selection on which it was called. This is a very powerful feature and you will see it being used by many professional libraries. Consider the following example:

```
$( '#button_submit' )
  .click(function() {
    $( this ).addClass( 'submit_clicked' );
  })
  .find( '#notification' )
    .attr( 'title', 'Message Sent' );x
```

In this snippet, we are chaining `click()`, `find()`, and `attr()` methods on a selector. Here, the `click()` method is executed, and once the execution finishes, the `find()` method locates the element with the `notification` ID and changes its `title` attribute to a string.

Traversal and manipulation

We discussed various methods of element selection using jQuery. We will discuss several DOM traversal and manipulation methods using jQuery in this section. These tasks would be rather tedious to achieve using native DOM manipulation. jQuery makes them intuitive and elegant.

Before we delve into these methods, let's familiarize ourselves with a bit of HTML terminology that we will be using from now on. Consider the following HTML:

```
<ul> <-This is the parent of both 'li' and ancestor of everything
  in
  <li> <-The first (li) is a child of the (ul)
    <span>  <-this is the descendent of the 'ul'
      <i>Hello</i>
    </span>
  </li>
  <li>World</li> <-both 'li' are siblings
</ul>
```

Using jQuery traversal methods, we select the first element and traverse through the DOM in relation to this element. As we traverse the DOM, we alter the original selection and we are either replacing the original selection with the new one or we are modifying the original selection.

For example, you can filter an existing selection to include only elements that match a certain criterion. Consider this example:

```
var list = $( 'li' ); //select all list elements
// filter items that has a class 'highlight' associated
var highlighted = list.filter( '.highlight );
// filter items that doesn't have class 'highlight' associated
var not_highlighted = list.not( '.highlight );
```

jQuery allows you to add and remove classes to elements. If you want to toggle class values for elements, you can use the `toggleClass()` method:

```
$( '#usename' ).addClass( 'hidden' );
$( '#usename' ).removeClass( 'hidden' );
$( '#usename' ).toggleClass( 'hidden' );
```

Most often, you may want to alter the value of elements. You can use the `val()` method to alter the form of element values. For example, the following line alters the value of all the `text` type inputs in the form:

```
$( 'input[type="text"]' ).val( 'Enter usename:' );
```

To modify element attributes, you can use the `attr()` method as follows:

```
$('a').attr( 'title', 'Click' );
```

jQuery has an incredible depth of functionality when it comes to DOM manipulation—the scope of this book restricts a detailed discussion of all the possibilities.

Working with browser events

When are you developing for browsers, you will have to deal with user interactions and events associated to them, for example, text typed in the textbox, scrolling of the page, mouse button press, and others. When the user does something on the page, an event takes place. Some events are not triggered by user interaction, for example, `load` event does not require a user input.

When you are dealing with mouse or keyboard events in the browser, you can't predict when and in which order these events will occur. You will have to constantly look for a key press or mouse move to happen. It's like running an endless background loop listening to some key or mouse event to happen. In traditional programming, this was known as polling. There were many variations of these where the waiting thread used to be optimized using queues; however, polling is still not a great idea in general.

Browsers provide a much better alternative to polling. Browsers provide you with programmatic means to react when an event occurs. These hooks are generally called listeners. You can register a listener that reacts to a particular event and executes an associated callback function when the event is triggered. Consider this example:

```
<script>
  addEventListener("click", function() {
    ...
  });
</script>
```

The `addEventListener` function registers its second argument as a callback function. This callback is executed when the event specified in the first argument is triggered.

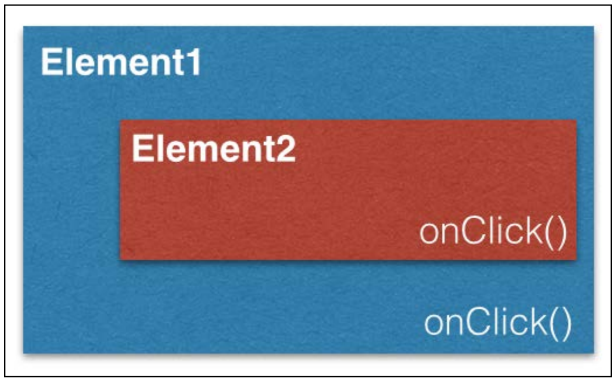
What we saw just now was a generic listener for the `click` event. Similarly, every DOM element has its own `addEventListener` method, which allows you to listen specifically on this element:

```
<button>Submit</button>
<p>No handler here.</p>
<script>
  var button = document.getElementById("#Bigbutton");
  button.addEventListener("click", function() {
    console.log("Button clicked.");
  });
</script>
```

In this example, we are using the reference to a specific element—a button with a `Bigbutton` ID—by calling `getElementById()`. On the reference of the button element, we are calling `addEventListener()` to assign a handler function for the click event. This is perfectly legitimate code that works fine in modern browsers such as Mozilla Firefox or Google Chrome. On Internet Explorer prior to IE9, however, this is not a valid code. This is because Microsoft implements its own custom `attachEvent()` method as opposed to the W3C standard `addEventListener()` prior to Internet Explorer 9. This is very unfortunate because you will have to write very bad hacks to handle browser-specific quirks.

Propagation

At this point, we should ask an important question—if an element and one of its ancestors have a handler on the same event, which handler will be fired first? Consider the following figure:



For example, we have **Element2** as a child of **Element1** and both have the `onClick` handler. When a user clicks on Element2, `onClick` on both Element2 and Element1 is triggered but the question is which one is triggered first. What should the event order be? Well, the answer, unfortunately, is that it depends entirely on the browser. When browsers first arrived, two opinions emerged, naturally, from Netscape and Microsoft.

Netscape decided that the first event triggered should be Element1's `onClick`. This event ordering is known as event capturing.

Microsoft decided that the first event triggered should be Element2's `onClick`. This event ordering is known as event bubbling.

These are two completely opposite views and implementations of how browsers handled events. To end this madness, **World Wide Web Consortium** (**W3C**) decided a wise middle path. In this model, an event is first captured until it reaches the target element and then bubbles up again. In this standard behavior, you can choose in which phase you want to register your event handler—either in the capturing or bubbling phase. If the last argument is true in `addEventListener()`, the event handler is set for the capturing phase, if it is false, the event handler is set for the bubbling phase.

There are times when you don't want the event to be raised by the parents if it was already raised by the child. You can call the `stopPropagation()` method on the event object to prevent handlers further up from receiving the event. Several events have a default action associated with them. For example, if you click on a URL link, you will be taken to the link's target. The JavaScript event handlers are called before the default behavior is performed. You can call the `preventDefault()` method on the event object to stop the default behavior from being triggered.

These are event basics when you are using plain JavaScript on a browser. There is a problem here. Browsers are notorious when it comes to defining event-handling behavior. We will look at jQuery's event handling. To make things easier to manage, jQuery always registers event handlers for the bubbling phase of the model. This means that the most specific elements will get the first opportunity to respond to any event.

jQuery event handling and propagation

jQuery event handling takes care of many of these browser quirks. You can focus on writing code that runs on most supported browsers. jQuery's support for browser events is simple and intuitive. For example, this code listens for a user to click on any button element on the page:

```
$('button').click(function(event) {
  console.log('Mouse button clicked');
});
```

Just like the `click()` method, there are several other helper methods to cover almost all kinds of browser event. The following helpers exist:

- `blur`
- `change`
- `click`
- `dblclick`

- `error`
- `focus`
- `keydown`
- `keypress`
- `keyup`
- `load`
- `mousedown`
- `mousemove`
- `mouseout`
- `mouseover`
- `mouseup`
- `resize`
- `scroll`
- `select`
- `submit`
- `unload`

Alternatively, you can use the `.on()` method. There are a few advantages of using the `on()` method as it gives you a lot more flexibility. The `on()` method allows you to bind a handler to multiple events. Using the `on()` method, you can work on custom events as well.

Event name is passed as the first parameter to the `on()` method just like the other methods that we saw:

```
$('button').on( 'click', function( event ) {
  console.log(' Mouse button clicked');
});
```

Once you've registered an event handler to an element, you can trigger this event as follows:

```
$('button').trigger( 'click' );
```

This event can also be triggered as follows:

```
$('button').click();
```

You can unbind an event using jQuery's `.off()` method. This will remove any event handlers that were bound to the specified event:

```
$('button').off( 'click' );
```

You can add more than one handler to an element:

```
$("#element")
.on("click", firstHandler)
.on("click", secondHandler);
```

When the event is fired, both the handlers will be invoked. If you want to remove only the first handler, you can use the `off()` method with the second parameter indicating the handler that you want to remove:

```
$("#element).off("click",firstHandler);
```

This is possible if you have the reference to the handler. If you are using anonymous functions as handlers, you can't get reference to them. In this case, you can use namespaced events. Consider the following example:

```
$("#element").on("click.firstclick",function() {
  console.log("first click");
});
```

Now that you have a namespaced event handler registered with the element, you can remove it as follows:

```
$("#element).off("click.firstclick");
```

A major advantage of using `.on()` is that you can bind to multiple events at once. The `.on()` method allows you to pass multiple events in a space-separated string. Consider the following example:

```
$('#inputBoxUserName').on('focus blur', function() {
  console.log( Handling Focus or blur event' );
});
```

You can add multiple event handlers for multiple events as follows:

```
$( "#heading" ).on({
  mouseenter: function() {
    console.log( "mouse entered on heading" );
  },
  mouseleave: function() {
    console.log( "mouse left heading" );
  },
  click: function() {
    console.log( "clicked on heading" );
  }
});
```

As of jQuery 1.7, all events are bound via the `on()` method, even if you call helper methods such as `click()`. Internally, jQuery maps these calls to the `on()` method. Due to this, it's generally recommended to use the `on()` method for consistency and faster execution.

Event delegation

Event delegation allows us to attach a single event listener to a parent element. This event will fire for all the descendants matching a selector even if these descendants will be created in the future (after the listener was bound to the element).

We discussed *event bubbling* earlier. Event delegation in jQuery works primarily due to event bubbling. Whenever an event occurs on a page, the event bubbles up from the element that it originated from, up to its parent, then up to the parent's parent, and so on, until it reaches the root element (`window`). Consider the following example:

```
<html>
  <body>
    <div id="container">
      <ul id="list">
        <li><a href="http://google.com">Google</a></li>
        <li><a href="http://myntra.com">Myntra</a></li>
        <li><a href="http://bing.com">Bing</a></li>
      </ul>
    </div>
  </body>
</html>
```

Now let's say that we want to perform some common action on any of the URL clicks. We can add an event handler to all the `a` elements in the list as follows:

```
$( "#list a" ).on( "click", function( event ) {
  console.log( $( this ).text() );
});
```

This works perfectly fine, but this code has a minor bug. What will happen if there is an additional URL added to the list as a result of some dynamic action? Let's say that we have an **Add** button that adds new URLs to this list. So, if the new list item is added with a new URL, the earlier event handler will not be attached to it. For example, if the following link is added to the list dynamically, clicking on it will not trigger the handler that we just added:

```
<li><a href="http://yahoo.com">Yahoo</a></li>
```

This is because such events are registered only when the `on()` method is called. In this case, as this new element did not exist when `.on()` was called, it does not get the event handler. With our understanding of event bubbling, we can visualize how the event will travel up the DOM tree. When any of the URLs are clicked on, the travel will be as follows:

```
a(click)->li->ul#list->div#container->body->html->root
```

We can create a delegated event as follows:

```
$( "#list" ).on( "click", "a", function( event ) {
  console.log( $( this ).text() );
});
```

We moved `a` from the original selector to the second parameter in the `on()` method. This second parameter of the `on()` method tells the handler to listen to this specific event and check whether the triggering element was the second parameter (the `a` in our case). As the second parameter matches, the handler function is executed. With this delegate event, we are attaching a single handler to the entire `ul#list`. This handler will listen to the click event triggered by any descendent of the `ul` element.

The event object

So far, we attached anonymous functions as event handlers. To make our event handlers more generic and useful, we can create named functions and assign them to the events. Consider the following lines:

```
function handlesClicks(event){
  //Handle click event
}
$("#bigButton").on('click', handlesClicks);
```

Here, we are passing a named function instead of an anonymous function to the `on()` method. Let's shift our focus now to the `event` parameter that we pass to the function. jQuery passes an event object with all the event callbacks. An event object contains very useful information about the event being triggered. In cases where we don't want the default behavior of the element to kick in, we can use the `preventDefault()` method of the event object. For example, we want to fire an AJAX request instead of a complete form submission or we want to prevent the default location to be opened when a URL anchor is clicked on. In these cases, you may also want to prevent the event from bubbling up the DOM. You can stop the event propagation by calling the `stopPropagation()` method of the event object. Consider this example:

```
$( "#loginform" ).on( "submit", function( event ) {
  // Prevent the form's default submission.
  event.preventDefault();
  // Prevent event from bubbling up DOM tree, also stops any
    delegation
  event.stopPropagation();
});
```

Apart from the event object, you also get a reference to the DOM object on which the event was fired. This element can be referred by `$(this)`. Consider the following example:

```
$( "a" ).click(function( event ) {
  var anchor = $( this );
  if ( anchor.attr( "href" ).match( "google" ) ) {
    event.preventDefault();
  }
});
```

Summary

This chapter was all about understanding JavaScript in its most important role—that of browser language. JavaScript plays the role of introducing dynamism on the web by facilitating DOM manipulation and event management on the browser. We discussed both of these concepts with and without jQuery. As the demands of the modern web are increasing, using libraries such as jQuery is essential. These libraries significantly improve the code quality and efficiency and, at the same time, give you the freedom to focus on important things.

We will focus on another incarnation of JavaScript—mainly on the server side. Node.js has become a popular JavaScript framework to write scalable server-side applications. We will take a detailed look at how we can best utilize Node.js for server applications.

9
Server-Side JavaScript

We have been focusing so far on the versatility of JavaScript as the language of the browser. It speaks volumes about the brilliance of the language given that JavaScript has gained significant popularity as a language to program scalable server systems. In this chapter, we will look at Node.js. Node.js is one of the most popular JavaScript frameworks used for server-side programming. Node.js is also one of the most watched project on GitHub and has superb community support.

Node uses V8, the virtual machine that powers Google Chrome, for server-side programming. V8 gives a huge performance benefit to Node because it directly compiles the JavaScript into native machine code over executing bytecode or using an interpreter as a middleware.

The versatility of V8 and JavaScript is a wonderful combination—the performance, reach, and overall popularity of JavaScript made Node an overnight success. In this chapter, we will cover the following topics:

- An asynchronous evented-model in a browser and Node.js
- Callbacks
- Timers
- EventEmitters
- Modules and npm

An asynchronous evented-model in a browser

Before we try to understand Node, let's try to understand JavaScript in a browser.

Node relies on event-driven and asynchronous platforms for server-side JavaScript. This is very similar to how browsers handle JavaScript. Both the browser and Node are event-driven and non-blocking when they use I/O.

To dive deeper into the event-driven and asynchronous nature of Node.js, let's first do a comparison of the various kinds of operations and costs associated with them:

L1 cache read	0.5 nanoseconds
L2 cache read	7 nanoseconds
RAM	100 nanoseconds
Read 4 KB randomly from SSD	150,000 ns
Read 1 MB sequentially from SSD	1,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns

These numbers are from `https://gist.github.com/jboner/2841832` and show how costly **Input/Output** (**I/O**) can get. The longest operations taken by a computer program are the I/O operations and these operations slow down the overall program execution if the program keeps waiting on these I/O operations to finish. Let's see an example of such an operation:

```
console.log("1");
var log = fileSystemReader.read("./verybigfile.txt");
console.log("2");
```

When you call `fileSystemReader.read()`, you are reading a file from the filesystem. As we just saw, I/O is the bottleneck here and can take quite a while before the read operation is completed. Depending on the kind of hardware, filesystem, OS, and so on, this operation will block the overall program execution quite a bit. The preceding code does some I/O that will be a blocking operation—the process will be blocked till I/O finishes and the data comes back. This is the traditional I/O model and most of us are familiar with this. However, this is costly and can cause terribly latency. Every process has associated memory and state—both these will be blocked till I/O is complete.

If a program blocks I/O, the Node server will refuse new requests. There are several ways of solving this problem. The most popular traditional approach is to use several threads to process requests—this technique is known as multithreading. If are you familiar with languages such as Java, chances are that you have written multithreaded code. Several languages support threads in various forms—a thread essentially holds its own memory and state. Writing multithreaded applications on a large scale is tough. When multiple threads are accessing a common shared memory or values, maintaining the correct state across these threads is a very difficult task. Threads are also costly when it comes to memory and CPU utilization. Threads that are used on synchronized resources may eventually get blocked.

The browser handles this differently. I/O in the browser happens outside the main execution thread and an event is emitted when I/O finishes. This event is handled by the callback function associated with that event. This type of I/O is non-blocking and asynchronous. As I/O is not blocking the main execution thread, the browser can continue to process other events as they come without waiting on any I/O. This is a powerful idea. Asynchronous I/O allows browsers to respond to several events and allows a high level of interactivity.

Node uses a similar idea for asynchronous processing. Node's event loop runs as a single thread. This means that the application that you write is essentially single-threaded. This does not mean that Node itself is single-threaded. Node uses **libuv** and is multithreaded—fortunately, these details are hidden within Node and you don't need to know them while developing your application.

Every call that involves an I/O call requires you to register a callback. Registering a callback is also asynchronous and returns immediately. As soon as an I/O operation is completed, its callback is pushed on the event loop. It is executed as soon as all the other callbacks that were pushed on the event loop before are executed. All operations are essentially thread-safe, primarily because there is no parallel execution path in the event loop that will require synchronization.

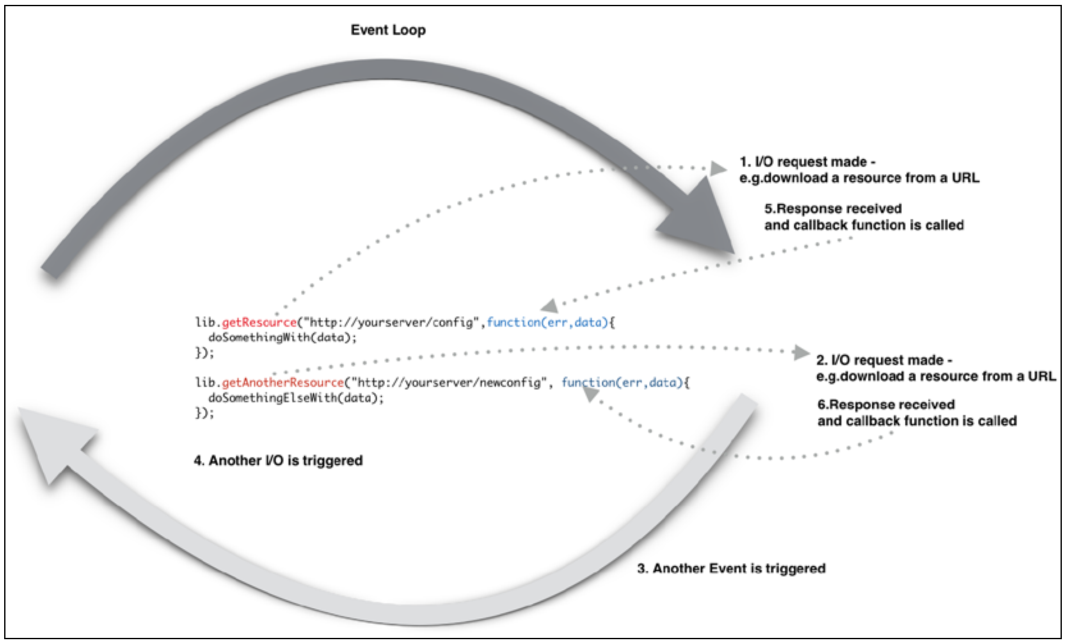
Essentially, there is only one thread running your code and there is no parallel execution; however, everything else except for your code runs in parallel.

Node.js relies on **libev** (`http://software.schmorp.de/pkg/libev.html`) to provide the event loop, which is supplemented by **libeio** (`http://software.schmorp.de/pkg/libeio.html`) that uses pooled threads to provide asynchronous I/O. To learn even more, take a look at the libev documentation at `http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod`.

Consider the following example of asynchronous code execution in Node.js:

```
var fs = require('fs');
console.log('1');
fs.readFile('./response.json', function (error, data) {
  if(!error){
    console.log(data);
  });
console.log('2');
```

In this program, we read the `response.json` file from the disk. When the disk I/O is finished, the callback is executed with parameters containing the argument's error, if any error occurred, and data, which is the file data. What you will see in the console is the output of `console.log('1')` and `console.log('2')` one immediately after another:


Event Loop
1. I/O request made -
e.g.download a resource from a URL
5.Response received
and callback function is called
2. I/O request made -
e.g.download a resource from a URL
6.Response received
and callback function is called
4. Another I/O is triggered
3. Another Event is triggered

```
lib.getRescurce("http://yourserver/config",function(err,data){
  doSomethingWith(data);
});

lib.getAnotherResource("http://yourserver/newconfig", function(err,data){
  doSomethingElseWith(data);
});
```

Node.js does not need any additional server component as it creates its own server process. A Node application is essentially a server running on a designated port. In Node, the server and application are the same.

Here is an example of a Node.js server responding with the **Hello Node** string when the `http://localhost:3000/` URL is run from a browser:

```
var http = require('http');
var server = http.createServer();
server.on('request', function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello Node\n');
});
server.listen(3000);
```

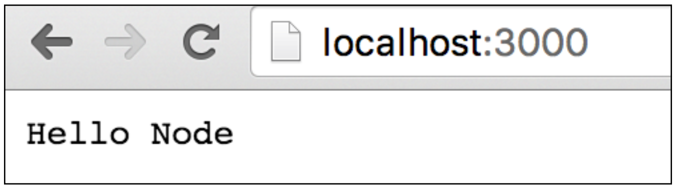
In this example, we are using an `http` module. If you recall our earlier discussions on the JavaScript module, you will realize that this is the CommonJS module implementation. Node has several modules compiled into the binary. The core modules are defined within Node's source. They can be located in the `lib/` folder.

They are loaded first if their identifier is passed to `require()`. For instance, `require('http')` will always return the built-in HTTP module, even if there is a file by this name.

After loading the module to handle HTTP requests, we create a `server` object and use a listener for a `request` event using the `server.on()` function. The callback is called whenever there is a request to this server on port `3000`. The callback receives `request` and `response` parameters. We are also setting the `Content-Type` header and HTTP response code before we send the response back. You can copy the preceding code, save it in a plain text file, and name it `app.js`. You can run the server from the command line using Node.js as follows:

```
$ » node app.js
```

Once the server is started, you can open the `http://localhost:3000` URL in a browser and you will be greeted with unexciting text:



```
Hello Node
```

If you want to inspect what's happening internally, you can issue a `curl` command as follows:

```
~ » curl -v http://localhost:3000
* Rebuilt URL to: http://localhost:3000/
*   Trying ::1...
* Connected to localhost (::1) port 3000 (#0)
> GET / HTTP/1.1
> Host: localhost:3000
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Date: Thu, 12 Nov 2015 05:31:44 GMT
< Connection: keep-alive
< Transfer-Encoding: chunked
<
Hello Node
* Connection #0 to host localhost left intact
```

Curl shows a nice request (>) and response (<) dialog including the request and response headers.

Callbacks

Callbacks in JavaScript usually take some time getting used to. If you are coming from some other non-asynchronous programming background, you will need to understand carefully how callbacks work; you may feel like you're learning programming for the first time. As everything is asynchronous in Node, you will be using callbacks for everything without trying to carefully structure them. The most important part of the Node.js project is sometimes the code organization and module management.

Callbacks are functions that are executed asynchronously at a later time. Instead of the code reading top to bottom procedurally, asynchronous programs may execute different functions at different times based on the order and speed that earlier functions such as HTTP requests or filesystem reads happen.

Whether a function execution is sequential or asynchronous depends on the context in which it is executed:

```
var i=0;
function add(num){
  console.log(i);
  i=i+num;
}
add(100);
console.log(i);
```

If you run this program using Node, you will see the following output (assuming that your file is named `app.js`):

```
~/Chapter9 » node app.js
0
100
```

This is what we are all used to. This is traditional synchronous code execution where each line is executed in a sequence. The code here defines a function and then on the next line calls this function, without waiting for anything. This is sequential control flow.

Things will be different if we introduced I/O to this sequence. If we try to read something from the file or call a remote endpoint, Node will execute these operations in an asynchronous fashion. For the next example, we are going to use a Node.js module called `request`. We will use this module to make HTTP calls. You can install the module as follows:

```
npm install request
```

We will discuss the use of npm later in this chapter. Consider the following example:

```
var request = require('request');
var status = undefined;
request('http://google.com', function (error, response, body) {
  if (!error && response.statusCode == 200) {
    status_code = response.statusCode;
  }
});
console.log(status);
```

When you execute this code, you will see that the value of the `status` variable is still `undefined`. In this example, we are making an HTTP call—this is an I/O operation. When we do an I/O operation, the execution becomes asynchronous. In the earlier example, we are doing everything within the memory and there was no I/O involved, hence, the execution was synchronous. When we run this program, all of the functions are immediately defined, but they don't all execute immediately. The `request()` function is called and the execution continues to the next line. If there is nothing to execute, Node will either wait for I/O to finish or it will exit. When the `request()` function finishes its work, it will execute the callback function (an anonymous function as the second parameter to the `request()` function). The reason that we got `undefined` in the preceding example is that nowhere in our code exists the logic that tells the `console.log()` statement to wait until the `request()` function has finished fetching the response from the HTTP call.

Callbacks are functions that get executed at some later time. This changes things in the way you organize your code. The idea around reorganizing the code is as follows:

- Wrapping the asynchronous code in a function
- Passing a callback function to the wrapper function

We will organize our previous example with these two ideas in mind. Consider this modified example:

```
var request = require('request');
var status = undefined;
function getSiteStatus(callback){
  request('http://google.com', function (error, response, body) {
    if (!error && response.statusCode == 200) {
      status_code = response.statusCode;
    }
    callback(status_code);
  });
}
function showStatusCode(status){
  console.log(status);
}
getSiteStatus(showStatusCode);
```

When you run this, you will get the following (correct) output:

```
$node app.js
200
```

What we changed was to wrap the asynchronous code in a `getSiteStatus()` function, pass a function named `callback()` as a parameter to this function, and execute this function on the last line of `getSiteStatus()`. The `showStatusCode()` callback function simply wraps around `console.log()` that we called earlier. The difference, however, is in the way the asynchronous execution works. The most important idea to understand while learning how to program with callbacks is that functions are first-class objects that can be stored in variables and passed around with different names. Giving simple and descriptive names to your variables is important in making your code readable by others. Now that the callback function is called once the HTTP call is completed, the value of the `status_code` variable will have a correct value. There are genuine circumstances where you want an asynchronous task executed only after another asynchronous task is completed. Consider this scenario:

```
http.createServer(function (req, res) {
  getURL(url, function (err, res) {
    getURLContent(res.data, function(err,res) {
      ...
    });
  });
});
```

As you can see, we are nesting one asynchronous function in another. This kind of nesting can result in code that is difficult to read and manage. This style of callback is sometimes known as **callback hell**. To avoid such a scenario, if you have code that has to wait for some other asynchronous code to finish, then you express that dependency by putting your code in functions that get passed around as callbacks. Another important idea is to name your functions instead of relying on anonymous functions as callbacks. We can restructure the preceding example into a more readable one as follows:

```
var urlContentProcessor = function(data){
  ...
}
var urlResponseProcessor = function(data){
  getURLContent(data,urlContentProcessor);
}
var createServer = function(req,res){
  getURL(url,urlResponseProcessor);
};
http.createServer(createServer);
```

This fragment uses two important concepts. First, we are using named functions and using them as callbacks. Second, we are not nesting these asynchronous functions. If you are accessing closure variables within the inner functions, the preceding would be a bit different implementation. In such cases, using inline anonymous functions is even more preferable.

Callbacks are most frequently used in Node. They are usually preferred to define logic for one-off responses. When you need to respond to repeating events, Node provides another mechanism for this. Before going further, we need to understand the function of timers and events in Node.

Timers

Timers are used to schedule the execution of a particular callback after a specific delay. There are two primary methods to set up such delayed execution: `setTimeout` and `setInterval`. The `setTimeout()` function is used to schedule the execution of a specific callback after a delay, while `setInterval` is used to schedule the repeated execution of a callback. The `setTimeout` function is useful to perform tasks that need to be scheduled such as housekeeping. Consider the following example:

```
setTimeout(function() {
  console.log("This is just one time delay");
},1000);
var count=0;
var t = setInterval(function() {
  count++;
  console.log(count);
  if (count> 5){
    clearInterval(t);
  }
}, 2000 );
```

First, we are using `setTimeout()` to execute a callback (the anonymous function) after a delay of 1,000 ms. This is just a one-time schedule for this callback. We scheduled the repeated execution of the callback using `setInterval()`. Note that we are assigning the value returned by `setInterval()` in a variable `t`—we can use this reference in `clearInterval()` to clear this schedule.

EventEmitters

We discussed earlier that callbacks are great for the execution of one-off logic. **EventEmitters** are useful in responding to repeating events. EventEmitters fire events and include the ability to handle these events when triggered. Several important Node APIs are built on EventEmitters.

Events raised by EventEmitters are handled through listeners. A listener is a callback function associated with an event—when the event fires, its associated listener is triggered as well. The `event.EventEmitter` is a class that is used to provide a consistent interface to emit (trigger) and bind callbacks to events.

As a common style convention, event names are represented by a camel-cased string; however, any valid string can be used as an event name.

Use `require('events')` to access the `EventEmitter` class:

```
var EventEmitter = require('events');
```

When an EventEmitter instance encounters an error, it emits an `error` event. Error events are treated as a special case in Node.js. If you don't handle these, the program exits with an exception stack.

All EventEmitters emit the `newListener` event when new listeners are added and `removeListener` when a listener is removed.

To understand the usage of EventEmitters, we will build a simplistic telnet server where different clients can log in and enter certain commands. Based on these commands, our server will respond accordingly:

```
var _net = require('net');
var _events = require ('events');
var _emitter = new events.EventEmitter();
_emitter.on('join', function(id,caller){
  console.log(id+" - joined");
});
_emitter.on('quit', function(id,caller){
  console.log(id+" - left");
});
```

```
var _server = _net.createServer(function(caller) {
  var process_id = caller.remoteAddress + ':' + caller.remotePort;
  _emitter.emit('join',id,caller);
  caller.on('end', function() {
    console.log("disconnected");
    _emitter.emit('quit',id,caller);
  });
});
_server.listen(8124);
```

In this code snippet, we are using the `net` module from Node. The idea here is to create a server and let the client connect to it via a standard `telnet` command. When a client connects, the server displays the client address and port, and when the client quits, the server logs this too.

When a client connects, we are emitting a `join` event, and when the client disconnects, we are emitting a `quit` event. We have listeners for both these events and they log appropriate messages on the server.

You start this program and connect to our server using telnet as follows:

```
telnet 127.0.0.1 8124
```

On the server console, you will see the server logging which client joined the server:

```
» node app.js
::ffff:127.0.0.1:51000 - joined
::ffff:127.0.0.1:51001 – joined
```

If any client quits the session, an appropriate message will appear as well.

Modules

When you are writing a lot of code, you soon reach a point where you have to start thinking about how you want to organize the code. Node modules are CommonJS modules that we discussed earlier when we discussed module patterns. Node modules can be published to the **Node Package Manager** (**npm**) repository. The npm repository is an online collection of Node modules.

Creating modules

Node modules can be either single files or directories containing one or more files. It's usually a good idea to create a separate module directory. The file in the module directory that will be evaluated is normally named `index.js`. A module directory can look as follows:

```
node_project/src/nav
                --- >index.js
```

In your project directory, the `nav` module directory contains the module code. Conventionally, your module code needs to reside in the `index.js` file—you can change this to another file if you want. Consider this trivial module called `geo.js`:

```
exports.area = function (r) {
  return 3.14 * r * r;
};
exports.circumference = function (r) {
  return 3.14 * 3.14 * r;
};
```

You are exporting two functions via `exports`. You can use the module using the `require` function. This function takes the name of the module or system path to the module's code. You can use the module that we created as follows:

```
var geo = require('./geo.js');
console.log(geo.area(2));
```

As we are exporting only two functions to the outside world, everything else remains private. If you recollect, we discussed the module pattern in detail—Node uses CommonJS modules. There is an alternative syntax to create modules as well. You can use `modules.exports` to export your modules. Indeed, `exports` is a helper created for `modules.exports`. When you use `exports`, it attaches the exported properties of a module to `modules.exports`. However, if `modules.exports` already has some properties attached to it, properties attached by `exports` are ignored.

The geo module created earlier in this section can be rewritten in order to return a single Geo constructor function rather than an object containing functions. We can rewrite the geo module and its usage as follows:

```
var Geo = function(PI) {
  this.PI = PI;
}
Geo.prototype.area = function (r) {
  return this.PI * r * r;
};
Geo.prototype.circumference = function (r) {
  return this.PI * this.PI * r;
};
module.exports = Geo;
```

Consider a `config.js` module:

```
var db_config = {
  server: "0.0.0.0",
  port: "3306",
  user: "mysql",
  password: "mysql"
};
module.exports = db_config;
```

If you want to access `db_config` from outside this module, you can use `require()` to include the module and refer the object as follows:

```
var config = require('./config.js');
console.log(config.user);
```

There are three ways to organize modules:

- Using a relative path, for example, `config = require('./lib/config.js')`
- Using an absolute path, for example, `config = require('/nodeproject/lib/config.js')`
- Using a module search, for example, `config = require('config')`

The first two are self-explanatory—they allow Node to look for a module in a particular location in the filesystem.

When you use the third option, you are asking Node to locate the module using the standard look method. To locate the module, Node starts at the current directory and appends `./node_modules/` to it. Node then attempts to load the module from this location. If the module is not found, then the search starts from the parent directory until the root of the filesystem is reached.

For example, if `require('config')` is called in `/projects/node/`, the following locations will be searched until a match a found:

- `/projects/node /node_modules/config.js`
- `/projects/node_modules/config.js`
- `/node_modules/config.js`

For modules downloaded from npm, using this method is relatively simple. As we discussed earlier, you can organize your modules in directories as long as you provide a point of entry for Node.

The easiest way to do this is to create the `./node_modules/supermodule/` directory, and insert an `index.js` file in this directory. The `index.js` file will be loaded by default. Alternatively, you can put a `package.json` file in the `mymodulename` folder, specifying the name and main file of the module:

```
{
  "name": "supermodule",
  "main": "./lib/config.js"
}
```

You have to understand that Node caches modules as objects. If you have two (or more) files requiring a specific module, the first `require` will cache the module in memory so that the second `require` will not have to reload the module source code. However, the second `require` can alter the module functionality if it wishes to. This is commonly called **monkey patching** and is used to modify a module behavior without really modifying or versioning the original module.

npm

The npm is the package manager used by Node to distribute modules. The npm can be used to install, update, and manage modules. Package managers are popular in other languages such as Python. The npm automatically resolves and updates dependencies for a package and hence makes your life easy.

Installing packages

There are two ways to install npm packages: locally or globally. If you want to use the module's functionality only for a specific Node project, you can install it locally relative to the project, which is default behavior of `npm install`. Alternatively, there are several modules that you can use as a command-line tool; in this case, you can install them globally:

```
npm install request
```

The `install` directive with `npm` will install a particular module—`request` in this case. To confirm that `npm install` worked correctly, check to see whether a `node_modules` directory exists and verify that it contains a directory for the package(s) that you installed.

As you start adding modules to your project, it becomes difficult to manage the version/dependency of each module. The best way to manage locally installed packages is to create a `package.json` file in your project.

A `package.json` file can help you in the following ways:

- Defining versions of each module that you want to install. There are times when your project depends on a specific version of a module. In this case, your `package.json` helps you download and maintain the correct version dependency.
- Serving as a documentation of all the modules that your project needs.
- Deploying and packaging your application without worrying about managing dependencies every time you deploy the code.

You can create `package.json` by issuing the following command:

```
npm init
```

After answering basic questions about your project, a blank `package.json` is created with content similar to the following:

```
{
  "name": "chapter9",
  "version": "1.0.0",
  "description": "chapter9 sample project",
  "main": "app.js",
  "dependencies": {
    "request": "^2.65.0"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "Chapter9",
    "sample",
    "project"
  ],
  "author": "Ved Antani",
  "license": "MIT"
}
```

You can manually edit this file in a text editor. An important part of this file is the `dependencies` tag. To specify the packages that your project depends on, you need to list the packages you'd like to use in your `package.json` file. There are two types of packages that you can list:

- `dependencies`: These packages are required by your application in production
- `devDependencies`: These packages are needed only for development and testing (for example, using the **Jasmine node package**)

In the preceding example, you can see the following dependency:

```
"dependencies": {
  "request": "^2.65.0"
},
```

This means that the project is dependent on the `request` module.

> The version of the module is dependent on the semantic versioning rules—`https://docs.npmjs.com/getting-started/semantic-versioning`.

Once your `package.json` file is ready, you can simply use the `npm install` command to install all the modules for your projects automatically.

There is a cool trick that I love to use. While installing modules from the command line, we can add the `--save` flag to add that module's dependency to the `package.json` file automatically:

```
npm install async --save
npm WARN package.json chapter9@1.0.0 No repository field.
npm WARN package.json chapter9@1.0.0 No README data
async@1.5.0 node_modules/async
```

In the preceding command, we installed the `async` module with the normal `npm` command with a `--save` flag. There is a corresponding entry automatically created in `package.json`:

```
"dependencies": {
  "async": "^1.5.0",
  "request": "^2.65.0"
},
```

JavaScript performance

Like any other language, writing correct JavaScript code at scale is an involved task. As the language matures, several of the inherent problems are being taken care of. There are several exceptional libraries that aid in writing good quality code. For most serious systems, *good code* = *correct code* + *high performance code*. The demands of new-generation software systems are high on performance. In this section, we will discuss a few tools that you can use to analyze your JavaScript code and understand its performance metrics.

We will discuss the following two ideas in this section:

- Profiling: Timing various functions and operations during script-profiling helps in identifying areas where you can optimize your code
- Network performance: Examining the loading of network resources such as images, stylesheets, and scripts

JavaScript profiling

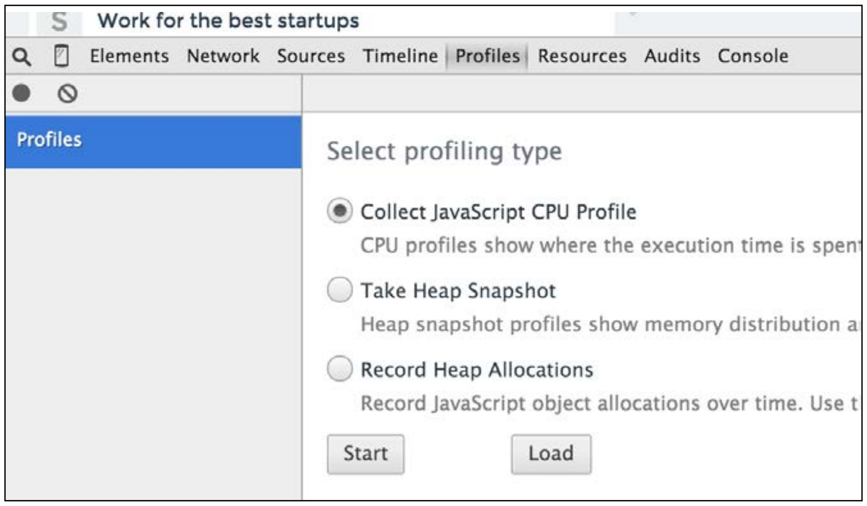
JavaScript profiling is critical to understand performance aspects of various parts of your code. You can observe timings of the functions and operations to understand which operation is taking more time. With this information, you can optimize the performance of time-consuming functions and tune the overall performance of your code. We will be focusing on the profiling options provided by Chrome's Developer Tools. There are comprehensive analysis tools that you can use to understand the performance metrics of your code.

The CPU profile

The CPU profile shows the execution time spent by various parts of your code. We have to inform DevTools to record the CPU profile data. Let's take the profiler for a spin.

You can enable the CPU profiler in DevTools as follows:

1. Open the Chrome DevTools **Profiles** panel.
2. Verify that **Collect JavaScript CPU Profile** is selected:


Work for the best startups
Elements Network Sources Timeline Profiles Resources Audits Console
Profiles
Start
Load

Select profiling type

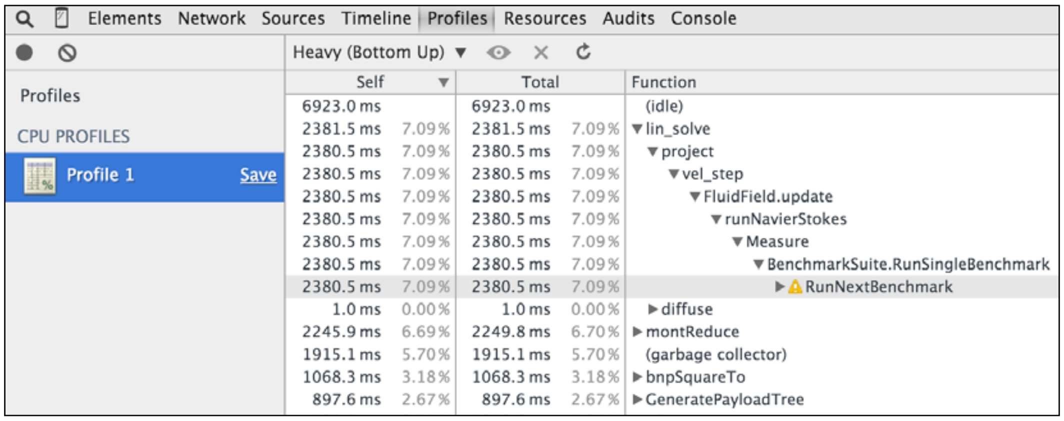
(●) Collect JavaScript CPU Profile
CPU profiles show where the execution time is spen

( ) Take Heap Snapshot
Heap snapshot profiles show memory distribution a

( ) Record Heap Allocations
Record JavaScript object allocations over time. Use t

For this chapter, we will be using Google's own benchmark page, `http://octane-benchmark.googlecode.com/svn/latest/index.html`. We will use this because it contains sample functions where we can see various performance bottlenecks and benchmarks. To start recording the CPU profile, open DevTools in Chrome, and in the **Profiles** tab, click on the **Start** button or press *Cmd*/*Ctrl* + *E*. Refresh the **V8 Benchmark Suite** page. When the page has completed reloading, a score for the benchmark tests is shown. Return to the **Profiles** panel and stop the recording by clicking on the **Stop** button or pressing *Cmd*/*Ctrl* + *E* again.

The recorded CPU profile shows you a detailed view of the functions and the execution time taken by them in the bottom-up fashion, as shown in the following image:

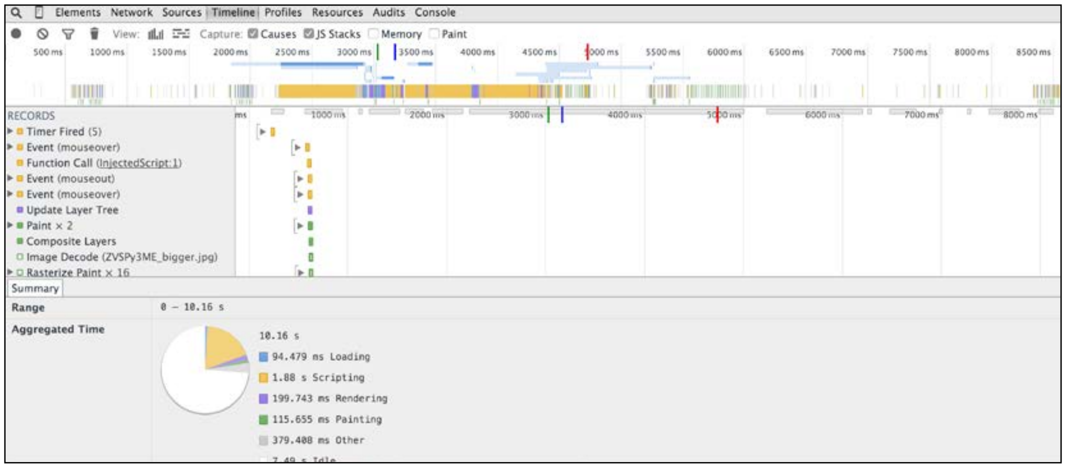


Self		Total		Function
6923.0 ms		6923.0 ms		(idle)
2381.5 ms	7.09%	2381.5 ms	7.09%	▼lin_solve
2380.5 ms	7.09%	2380.5 ms	7.09%	▼project
2380.5 ms	7.09%	2380.5 ms	7.09%	▼vel_step
2380.5 ms	7.09%	2380.5 ms	7.09%	▼FluidField.update
2380.5 ms	7.09%	2380.5 ms	7.09%	▼runNavierStokes
2380.5 ms	7.09%	2380.5 ms	7.09%	▼Measure
2380.5 ms	7.09%	2380.5 ms	7.09%	▼BenchmarkSuite.RunSingleBenchmark
2380.5 ms	7.09%	2380.5 ms	7.09%	▶⚠RunNextBenchmark
1.0 ms	0.00%	1.0 ms	0.00%	▶diffuse
2245.9 ms	6.69%	2249.8 ms	6.70%	▶montReduce
1915.1 ms	5.70%	1915.1 ms	5.70%	(garbage collector)
1068.3 ms	3.18%	1068.3 ms	3.18%	▶bnpSquareTo
897.6 ms	2.67%	897.6 ms	2.67%	▶GeneratePayloadTree

The Timeline view

The Chrome DevTools **Timeline** tool is the first place you can start looking at the overall performance of your code. It lets you record and analyze all the activity in your application as it runs.

The **Timeline** provides you with a complete overview of where time is spent when loading and using your site. A timeline recording includes a record for each event that occurred and is displayed in a **waterfall** graph:

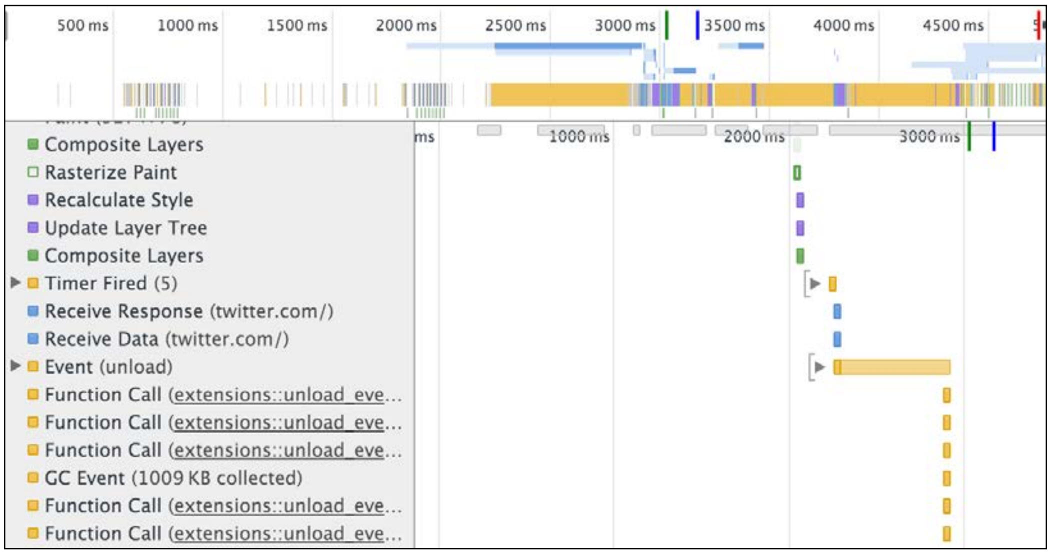


RECORDS
- Timer Fired (5)
- Event (mouseover)
- Function Call (InjectedScript:1)
- Event (mouseout)
- Event (mouseover)
- Update Layer Tree
- Paint × 2
- Composite Layers
- Image Decode (ZVSPy3ME_bigger.jpg)
- Rasterize Paint × 16

Range	0 – 10.16 s
Aggregated Time	10.16 s

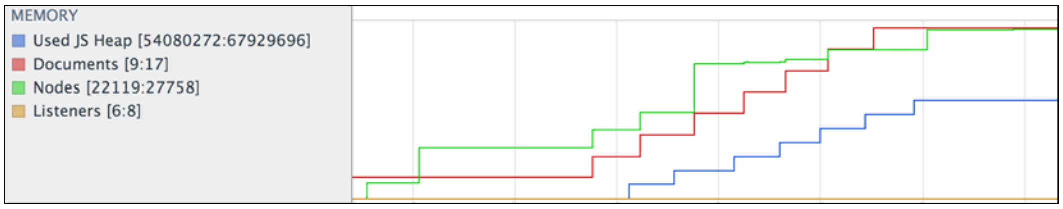
```
10.16 s
94.479 ms Loading
1.88 s Scripting
199.743 ms Rendering
115.655 ms Painting
379.408 ms Other
7.49 s Idle
```

The preceding screen shows you the timeline view when we try to render `https://twitter.com/` in the browser. The timeline view gives you an overall view of which operation took how much time in execution:



- 🟩 Composite Layers
- ☐ Rasterize Paint
- 🟪 Recalculate Style
- 🟪 Update Layer Tree
- 🟩 Composite Layers
- ▶ 🟨 Timer Fired (5)
- 🟦 Receive Response (twitter.com/)
- 🟦 Receive Data (twitter.com/)
- ▶ 🟨 Event (unload)
- 🟨 Function Call (extensions::unload_eve...
- 🟨 Function Call (extensions::unload_eve...
- 🟨 Function Call (extensions::unload_eve...
- 🟨 GC Event (1009 KB collected)
- 🟨 Function Call (extensions::unload_eve...
- 🟨 Function Call (extensions::unload_eve...

In the preceding screenshot, we can see the progressive execution of various JavaScript functions, network calls, resource downloads, and other operations involved in rendering the Twitter home page. This view gives us a very good idea about which operations may be taking longer. Once we identify such operations, we can optimize them for performance. The **Memory** view is a great tool to understand how the memory is used during the lifetime of your application in the browser. The **Memory** view shows you a graph of the memory used by your application over time and maintains a counter of the number of documents, DOM nodes, and event listeners that are held in the memory. The **Memory** view can help detect memory leaks and give you good enough hints to understand what optimizations are required:



MEMORY

- 🟦 Used JS Heap [54080272:67929696]
- 🟥 Documents [9:17]
- 🟩 Nodes [22119:27758]
- 🟨 Listeners [6:8]

JavaScript performance is a fascinating subject and deserves its own dedicated text. I would urge you to explore Chrome's DevTools and understand how best to use the tools to detect and diagnose performance problems in your code.

Summary

In this chapter, we looked at a different avatar of JavaScript—that of a server-side framework in the form of Node.js.

Node offers an asynchronous evented-model to program scalable and high-performance server applications in JavaScript. We dived deep into some core concepts on Node, such as an event loop, callbacks, modules, and timers. Understanding them is critical to write good Node code. We also discussed several techniques to structure Node code and callbacks in a better way.

With this, we reach the conclusion of our exploration of a brilliant programming language. JavaScript has been instrumental in the evolution of the World Wide Web because of its sheer versatility. The language continues to expand its horizons and improves with each new iteration.

We started our journey with understanding the building blocks of the grammar and syntax of the language. We grasped the fundamental ideas of closures and the functional behavior of JavaScript. These concepts are so essential that most of the JavaScript patterns are based on them. We looked at how we can utilize these patterns to write better code with JavaScript. We studied how JavaScript can operate on a DOM and how to use jQuery to manipulate the DOM effectively. Finally, we looked at the server-side avatar of JavaScript in Node.js.

This book should have enabled you to think differently when you start programming in JavaScript. Not only will you think about common patterns when you code, but also appreciate and use newer language features by ES6.

Module 2

Learning Object-Oriented Programming

Explore and crack the OOP code in Python, JavaScript, and C#

1
Objects Everywhere

Objects are everywhere, and therefore, it is very important to recognize elements, known as objects, from real-world situations. It is also important to understand how they can easily be translated into object-oriented code. In this chapter, you will learn the principles of object-oriented paradigms and some of the differences in the approaches towards object-oriented code in each of the three programming languages: Python, JavaScript, and C#. In this chapter, we will:

- Understand how real-world objects can become a part of fundamental elements in the code
- Recognize objects from nouns
- Generate blueprints for objects and understand classes
- Recognize attributes to generate fields
- Recognize actions from verbs to generate methods
- Work with UML diagrams and translate them into object-oriented code
- Organize blueprints to generate different classes
- Identify the object-oriented approaches in Python, JavaScript, and C#

Recognizing objects from nouns

Let's imagine, we have to develop a new simple application, and we receive a description with the requirements. The application must allow users to calculate the areas and perimeters of squares, rectangles, circles, and ellipses.

It is indeed a very simple application, and you can start writing code in Python, JavaScript, and C#. You can create four functions that calculate the areas of the shapes mentioned earlier. Moreover, you can create four additional functions that calculate the perimeters for them. For example, the following seven functions would do the job:

- `calculateSquareArea`: This receives the parameters of the square and returns the value of the calculated area for the shape
- `calculateRectangleArea`: This receives the parameters of the rectangle and returns the value of the calculated area for the shape
- `calculateCircleArea`: This receives the parameters of the circle and returns the value of the calculated area for the shape
- `calculateEllipseArea`: This receives the parameters of the ellipse and returns the value of the calculated area for the shape
- `calculateSquarePerimeter`: This receives the parameters of the square and returns the value of the calculated perimeter for the shape
- `calculateRectanglePerimeter`: This receives the parameters of the rectangle and returns the value of the calculated perimeter for the shape
- `calculateCirclePerimeter`: This receives the parameters of the circle and returns the value of the calculated perimeter for the shape

However, let's forget a bit about programming languages and functions. Let's recognize the real-world objects from the application's requirements. It is necessary to calculate the areas and perimeters of four elements, that is, four nouns in the requirements that represent real-life objects:

- Square
- Rectangle
- Circle
- Ellipse

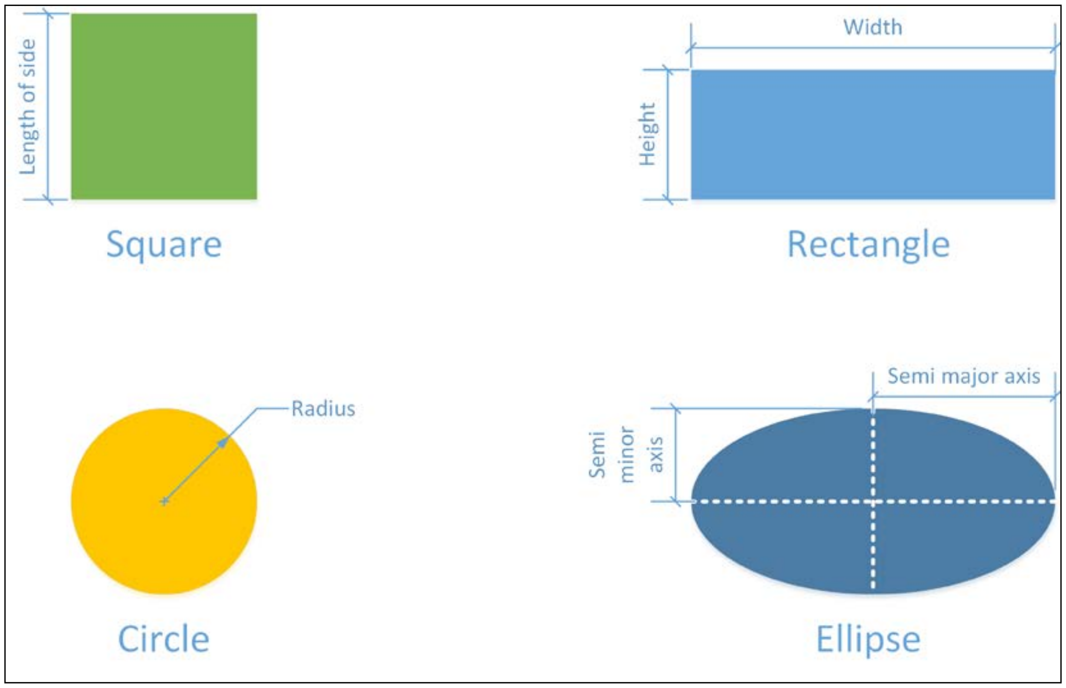
We can design our application by following an object-oriented paradigm. Instead of creating a set of functions that perform the required tasks, we can create software objects that represent the state and behavior of a square, rectangle, circle, and an ellipse. This way, the different objects mimic the real-world shapes. We can work with the objects to specify the different attributes required to calculate their areas and their perimeters.

Now, let's move to the real world and think about the four shapes. Imagine that you have to draw the four shapes on paper and calculate both their areas and perimeters. What information do you require for each of the shapes? Think about this, and then, take a look at the following table that summarizes the data required for each shape:

Shape	**Required data**
Square	Length of side
Rectangle	Width and height
Circle	Radius (usually labeled as *r*)
Ellipse	Semi-major axis (usually labeled as *a*) and semi-minor axis (usually labeled as *b*)

The data required by each of the shapes is going to be encapsulated in each object. For example, the object that represents a rectangle encapsulates both the rectangle's width and height. *Data encapsulation* is one of the major pillars of object-oriented programming.

The following diagram shows the four shapes drawn and their elements:



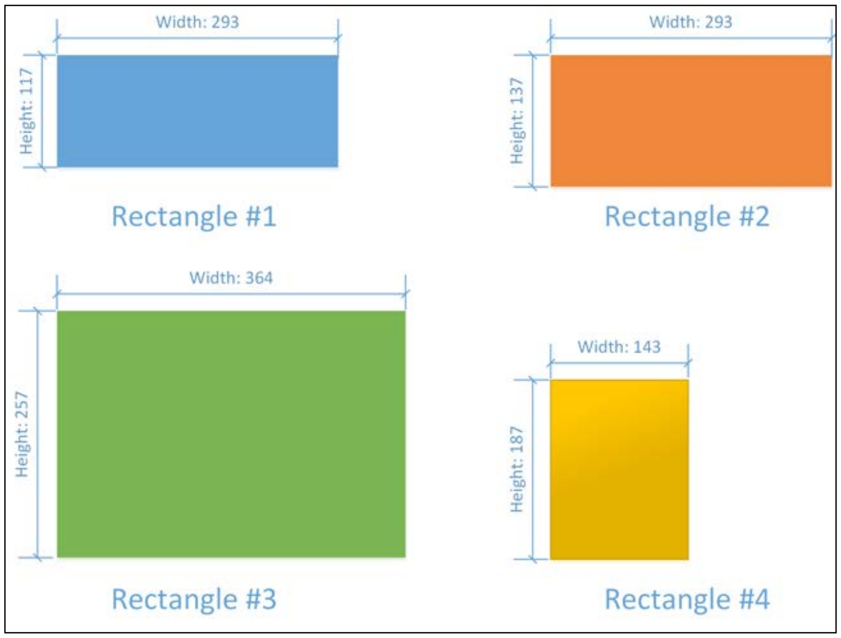
Generating blueprints for objects

Imagine that you want to draw and calculate the areas of four different rectangles. You will end up with four rectangles drawn, with their different widths, heights, and calculated areas. It would be great to have a blueprint to simplify the process of drawing each rectangle with their different widths and heights.

In object-oriented programming, a class is a blueprint or a template definition from which the objects are created. Classes are models that define the state and behavior of an object. After defining a class that defines the state and behavior of a rectangle, we can use it to generate objects that represent the state and behavior of each real-world rectangle.

Objects are also known as instances. For example, we can say each rectangle object is an instance of the rectangle class.

The following image shows four rectangle instances drawn, with their widths and heights specified: Rectangle #1, Rectangle #2, Rectangle #3, and Rectangle #4. We can use a `rectangle` class as a blueprint to generate the four different `rectangle` instances. It is very important to understand the difference between a class and the objects or instances generated through its usage. Object-oriented programming allows us to discover the blueprint we used to generate a specific object. Thus, we are able to infer that each object is an instance of the `rectangle` class.



We recognized four completely different real-world objects from the application's requirements. We need classes to create the objects, and therefore, we require the following four classes:

- Square
- Rectangle
- Circle
- Ellipse

Recognizing attributes/fields

We already know the information required for each of the shapes. Now, it is time to design the classes to include the necessary attributes that provide the required data to each instance. In other words, we have to make sure that each class has the necessary variables that encapsulate all the data required by the objects to perform all the tasks.

Let's start with the **`Square`** class. It is necessary to know the length of side for each instance of this class, that is, for each `square` object. Thus, we need an encapsulated variable that allows each instance of this class to specify the value of the length of side.

> The variables defined in a class to encapsulate data for each instance of the class are known as **attributes** or **fields**. Each instance has its own independent value for the attributes or fields defined in the class.

The `Square` class defines a floating point attribute named `LengthOfSide` whose initial value is equal to `0` for any new instance of the class. After you create an instance of the `Square` class, it is possible to change the value of the `LengthOfSide` attribute.

For example, imagine that you create two instances of the `Square` class. One of the instances is named **`square1`**, and the other is **`square2`**. The instance names allow you to access the encapsulated data for each object, and therefore, you can use them to change the values of the exposed attributes.

Imagine that our object-oriented programming language uses a dot (`.`) to allow us to access the attributes of the instances. So, `square1.LengthOfSide` provides access to the length of side for the `Square` instance named `square1`, and `square2.LengthOfSide` does the same for the `Square` instance named `square2`.

You can assign the value `10` to `square1.LengthOfSide` and `20` to `square2.LengthOfSide`. This way, each `Square` instance is going to have a different value for the `LengthOfSide` attribute.

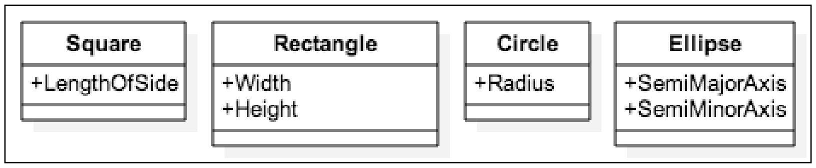
Now, let's move to the **`Rectangle`** class. We can define two floating-point attributes for this class: `Width` and `Height`. Their initial values are also going to be `0`. Then, you can create two instances of the `Rectangle` class: `rectangle1` and **`rectangle2`**.

You can assign the value `10` to `rectangle1.Width` and `20` to `rectangle1.Height`. This way, `rectangle1` represents a 10 x 20 rectangle. You can assign the value `30` to `rectangle2.Width` and `50` to `rectangle2.Height` to make the second `Rectangle` instance, which represents a 30 x 50 rectangle.

The following table summarizes the floating-point attributes defined for each class:

Class name	**Attributes list**
Square	LengthOfSide
Rectangle	Width Height
Circle	Radius
Ellipse	SemiMajorAxis

The following image shows a **UML** (**Unified Modeling Language**) diagram with the four classes and their attributes:



Recognizing actions from verbs – methods

So far, we have designed four classes and identified the necessary attributes for each of them. Now, it is time to add the necessary pieces of code that work with the previously defined attributes to perform all the tasks. In other words, we have to make sure that each class has the necessary encapsulated functions that process the attribute values specified in the objects to perform all the tasks.

Let's start with the **`Square`** class. The application's requirements specified that we have to calculate the areas and perimeters of squares. Thus, we need pieces of code that allow each instance of this class to use the **`LengthOfSide`** value to calculate the area and the perimeter.

> The functions or subroutines defined in a class to encapsulate the behavior for each instance of the class are known as **methods**. Each instance can access the set of methods exposed by the class. The code specified in a method is able to work with the attributes specified in the class. When we execute a method, it will use the attributes of the specific instance. A good practice is to define the methods in a logical place, that is, in the place where the required data is kept.

The `Square` class defines the following two parameterless methods. Notice that we declare the code for both methods in the definition of the `Square` class:

- `CalculateArea`: This returns a floating-point value with the calculated area for the square. The method returns the square of the `LengthOfSide` attribute value (*$LengthOfSide^2$* or *LengthOfSide ^ 2*).
- `CalculatePerimeter`: This returns a floating-point value with the calculated perimeter for the square. The method returns the `LengthOfSide` attribute value multiplied by `4` (*4 * LengthOfSide*).

Imagine that, our object-oriented programming language uses a dot (`.`) to allow us to execute methods of the instances. Remember that we had two instances of the `Square` class: `square1` with `LengthOfSide` equal to `10` and `square2` with `LengthOfSide` equal to `20`. If we call `square1.CalculateArea`, it would return the result of *10^2*, which is `100`. On the other hand, if we call `square2.CalculateArea`, it would return the result of *20^2*, which is `400`. Each instance has a diverse value for the `LengthOfSide` attribute, and therefore, the results of executing the `CalculateArea` method are different.

If we call `square1.CalculatePerimeter`, it would return the result of *4 * 10*, which is `40`. On the other hand, if we call `square2.CalculatePerimeter`, it would return the result of *4 * 20*, which is `80`.

Now, let's move to the **`Rectangle`** class. We need exactly two methods with the same names specified for the `Square` class. However, they have to calculate the results in a different way.

- `CalculateArea`: This returns a floating-point value with the calculated area for the rectangle. The method returns the result of the multiplication of the `Width` attribute value by the `Height` attribute value (*Width * Height*).
- `CalculatePerimeter`: This returns a floating-point value with the calculated perimeter for the rectangle. The method returns the sum of two times the `Width` attribute value and two times the `Height` attribute value (*2 * Width + 2 * Height*).

Remember that, we had two instances of the `Rectangle` class: `rectangle1` representing a 10 x 20 rectangle and `rectangle2` representing a 30 x 50 rectangle. If we call `rectangle1.CalculateArea`, it would return the result of *10 * 20*, which is `200`. On the other hand, if we call `rectangle2.CalculateArea`, it would return the result of *30 * 50*, which is `1500`. Each instance has a diverse value for both the `Width` and `Height` attributes, and therefore, the results of executing the `CalculateArea` method are different.

If we call `rectangle1.CalculatePerimeter`, it would return the result of *2 * 10 + 2 * 20*, which is `60`. On the other hand, if we call `rectangle2. CalculatePerimeter`, it would return the result of *2 * 30 + 2 * 50*, which is `160`.

The **`Circle`** class also needs two methods with the same names. The two methods are explained as follows:

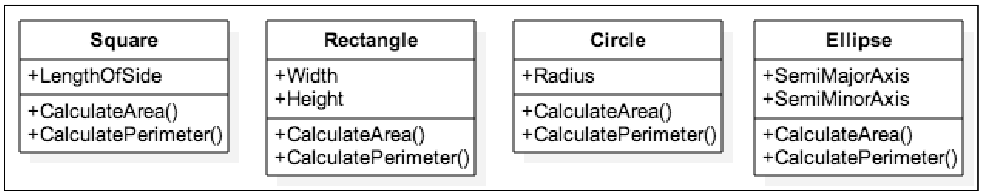
- `CalculateArea`: This returns a floating-point value with the calculated area for the circle. The method returns the result of the multiplication of π by the square of the `Radius` attribute value ($\pi * Radius^2$ or $\pi * (Radius \wedge 2)$).
- `CalculatePerimeter`: This returns a floating-point value with the calculated perimeter for the circle. The method returns the result of the multiplication of π by two times the `Radius` attribute value.

Finally, the **`Ellipse`** class defines two methods with the same names but with different code and a specific problem with the perimeter. The following are the two methods:

- `CalculateArea`: This returns a floating-point value with the calculated area for the ellipse. The method returns the result of the multiplication of π by the square of the `Radius` attribute value ($\pi * SemiMajorAxis * SemiMinorAxis$).

- `CalculatePerimeter`: This returns a floating-point value with the calculated approximation of the perimeter for the ellipse. Perimeters are very difficult to calculate for ellipses, and therefore, there are many formulas that provide approximations. An exact formula needs an infinite series of calculations. Thus, let's consider that the method returns the result of a formula that isn't very accurate and that we will have to improve on it later. The method returns the result of $2 * \pi * SquareRoot\ ((SemiMajorAxis^2 + SemiMinorAxis^2) / 2)$.

The following figure shows an updated version of the UML diagram with the four classes, their attributes, and their methods:



Organizing the blueprints – classes

So far, our object-oriented solution includes four classes with their attributes and methods. However, if we take another look at these four classes, we would notice that all of them have the same two methods: `CalculateArea` and `CalculatePerimeter`. The code for the methods in each class is different, because each shape uses a different formula to calculate either the area or the perimeter. However, the declarations or the contracts for the methods are the same. Both methods have the same name, are always parameterless, and both return a floating-point value.

When we talked about the four classes, we said we were talking about four different geometrical shapes or simply, shapes. Thus, we can generalize the required behavior for the four shapes. The four shapes must declare the `CalculateArea` and `CalculatePerimeter` methods with the previously explained declarations. We can create a contract to make sure that the four classes provide the required behavior.

The contract will be a class named `Shape`, and it will generalize the requirements for the geometrical shapes in our application. The `Shape` class declares two parameterless methods that return a floating-point value: `CalculateArea` and `CalculatePerimeter`. Then, we can declare the four classes as subclasses of the **`Shape`** class that inherit these definitions, but provide the specific code for each of these methods.

We can define the `Shape` class as an abstract class, because we don't want to be able to create instances of this class. We want to be able to create instances of `Square`, `Rectangle`, `Circle`, or `Ellipse`. In this case, the `Shape` abstract class declares two abstract methods. We call `CalculateArea` and `CalculatePerimeter` abstract methods because the abstract class declares them without an implementation, that is, without code. The subclasses of `Shape` implement the methods because they provide code while maintaining the same method declarations specified in the `Shape` superclass. *Abstraction* and *hierarchy* are the two major pillars of object-oriented programming.

Object-oriented programming allows us to discover whether an object is an instance of a specific superclass. After we changed the organization of the four classes and they became subclasses of the `Shape` class, any instance of `Square`, `Rectangle`, `Circle`, or `Ellipse` is also an instance of the `Shape` class. In fact, it isn't difficult to explain the abstraction because we are telling the truth about the object-oriented model that represents the real world. It makes sense to say that a rectangle is indeed a shape, and therefore, an instance of a `Rectangle` class is a `Shape` class. An instance of a `Rectangle` class is both a `Shape` class (the superclass of the `Rectangle` class) and a `Rectangle` class (the class that we used to create the object).

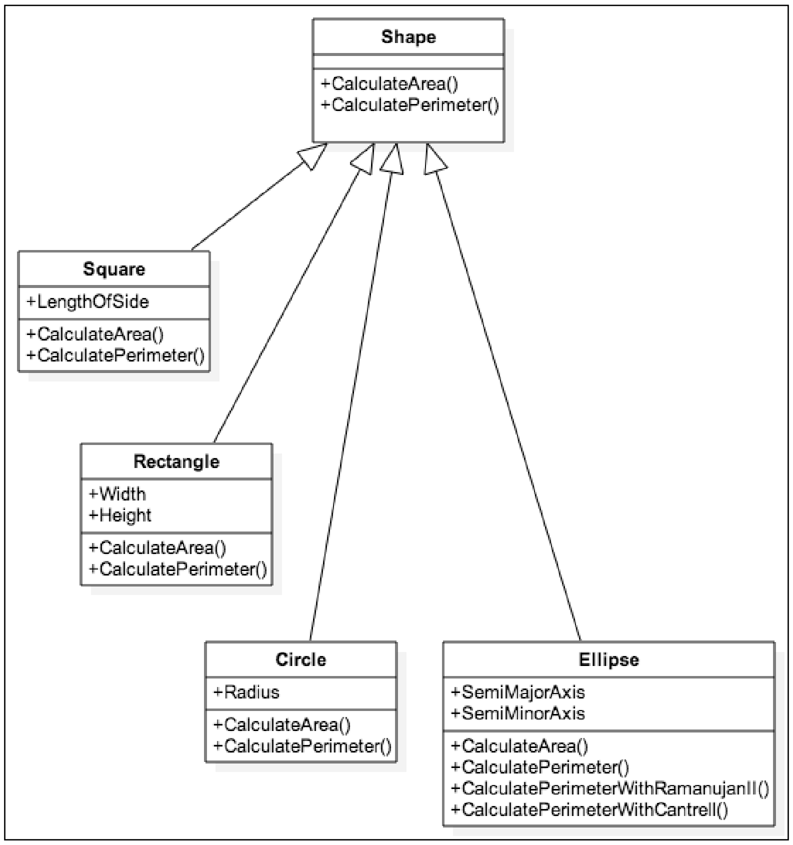
When we were implementing the `Ellipse` class, we discovered a specific problem for this shape; there are many formulas that provide approximations of the perimeter value. Thus, it makes sense to add additional methods that calculate the perimeter using other formulas.

We can define the following two additional parameterless methods, that is, two methods without any parameter. These methods return a floating-point value to the `Ellipse` class to solve the specific problem of the ellipse shape. The following are the two methods:

- `CalculatePerimeterWithRamanujanII`: This uses the second version of a formula developed by Srinivasa Aiyangar Ramanujan
- `CalculatePerimeterWithCantrell`: This uses a formula proposed by David W. Cantrell

This way, the `Ellipse` class implements the methods specified in the `Shape` superclass. The `Ellipse` class also adds two specific methods that aren't included in any of the other subclasses of `Shape`.

The following diagram shows an updated version of the UML diagram with the abstract class, its four subclasses, their attributes, and their methods:



Object-oriented approaches in Python, JavaScript, and C#

Python, JavaScript, and C# support object-oriented programming, also known as OOP. However, each programming language takes a different approach. Both Python and C# support classes and inheritance. Therefore, you can use the different syntax provided by each of these programming languages to declare the `Shape` class and its four subclasses. Then, you can create instances of each of the subclasses and call the different methods.

On the other hand, JavaScript uses an object-oriented model that doesn't use classes. This object-oriented model is known as **prototype-based programming**. However, don't worry. Everything you have learned so far in your simple object-oriented design journey can be coded in JavaScript. Instead of using inheritance to achieve behavior reuse, we can expand upon existing objects. Thus, we can say that objects serve as prototypes in JavaScript. Instead of focusing on classes, we work with instances and decorate them to emulate inheritance in class-based languages.

The object-oriented model known as prototype-based programing is also known by other names such as classless programming, instance-based programming, or prototype-oriented programming.

There are other important differences between Python, JavaScript, and C#. They have a great impact on the way you can code object-oriented designs. However, you will learn different ways throughout this book to make it possible to code the same object-oriented design in the three programming languages.

Summary

In this chapter, you learned how to recognize real-world elements and translate them into the different components of the object-oriented paradigm: classes, attributes, methods, and instances. You understood the differences between classes (blueprints or templates) and the objects (instances). We designed a few classes with attributes and methods that represented blueprints for real-life objects. Then, we improved the initial design by taking advantage of the power of abstraction, and we specialized in the `Ellipse` class.

Now that you have learned some of the basics of the object-oriented paradigm, you are ready to start creating classes and instances in Python, JavaScript, and C# in the next chapter.

2
Classes and Instances

In this chapter, we will start generating blueprints to create objects in each of the three programming languages: Python, JavaScript, and C#. We will:

- Understand the differences between classes, prototypes, and instances in object-oriented programming
- Learn an object's lifecycle and how object constructors and destructors work
- Declare classes in Python and C# and use workarounds to have a similar feature in JavaScript
- Customize the process that takes place when you create instances in Python, C#, and JavaScript
- Customize the process that takes place when you destroy instances in Python, C#, and JavaScript
- Create different types of objects in Python, C#, and JavaScript

Understanding classes and instances

In the previous chapter, you learned some of the basics of the object-oriented paradigm, including classes and objects, also known as instances. Now, when you dive deep into the programming languages, the class is always going to be the type and the blueprint. The object is the working instance of the class, and one or more variables can hold a reference to an instance.

Let's move to the world of our best friends, the dogs. If we want to model an object-oriented application that has to work with dogs and about a dozen dog breeds, we will definitely have a `Dog` abstract class. Each dog breed required in our application will be a subclass of the `Dog` superclass. For example, let's assume that we have the following subclasses of `Dog`:

- `TibetanSpaniel`: This is a blueprint for the dogs that belong to the Tibetan Spaniel breed
- `SmoothFoxTerrier`: This is a blueprint for the dogs that belong to the Smooth Fox Terrier breed

So, each dog breed will become a subclass of `Dog` and a type in the programming language. Each dog breed is a blueprint that we will be able to use to create instances. `Brian` and `Merlin` are two dogs. `Brian` belongs to the Tibetan Spaniel breed, and `Merlin` belongs to the Smooth Fox Terrier breed. In our application, `Brian` will be an instance of the `TibetanSpaniel` subclass, and `Merlin` will be an instance of the `SmoothFoxTerrier` subclass.

As both `Brian` and `Merlin` are dogs, they will share many attributes. Some of these attributes will be initialized by the class, because the dog breed they belong to determines some features, for example, the area of origin, the average size, and the watchdog ability. However, other attributes will be specific to the instance, such as the name, weight, age, and hair color.

Understanding constructors and destructors

When you ask the programming language to create an instance of a specific class, something happens under the hood. The programming language runtime creates a new instance of the specified type, allocates the necessary memory, and then executes the code specified in the constructor. When the runtime executes the code within the constructor, there is already a live instance of the class. Thus, you have access to the attributes and methods defined in the class. However, as you might have guessed, you must be careful with the code you put within the constructor, because you might end up generating large delays when you create instances of the class.

Constructors are extremely useful to execute setup code and properly initialize a new instance.

So, for example, before you can call the `CalculateArea` method, you want the `Width` and `Height` attributes for each new `Rectangle` instance to have a value initialized to `0`. Constructors are extremely useful when we want to define the attributes of the instances of a class right after their creation.

Sometimes, we need specific arguments to be available when we are creating an instance. We can design different constructors with the necessary arguments and use them to create instances of a class. This way, we can make sure that there is no way of creating specific classes without using the authorized constructors that ask for the necessary arguments.

At some point, your application won't need to work with an instance anymore. For example, once you calculate the perimeter of an ellipse and display the results to the user, you don't need the specific `Ellipse` instance anymore. Some programming languages require you to be careful about leaving live instances alive. You have to explicitly destroy them and de-allocate the memory that it was consuming.

The runtimes of Python, C#, and JavaScript use a garbage-collection mechanism that automatically de-allocates memory used by instances that aren't referenced anymore. The garbage-collection process is a bit more complicated, and each programming language and runtime has specific considerations that should be taken into account to avoid unnecessary memory pressure. However, let's keep our focus on the object's life cycle. In these programming languages, when the runtime detects that you aren't referencing an instance anymore and when a garbage collection occurs, the runtime executes the code specified within the instance's destructor.

[You can use the destructor to perform any necessary cleanup before the object is destroyed and removed from memory. However, take into account that JavaScript doesn't provide you with the possibility to customize a destructor.]

For example, think about the following situation. You need to count the number of instances of a specific class that are being kept alive. You can have a variable shared by all the classes. Then, you customize the class constructor to atomically increase the value for the counter, that is, increase the value of the variable shared by all the classes of the same time. Finally, you customize the class destructor to automatically decrease the value for the counter. This way, you can check the value of this variable to know the objects that are being referenced in your application.

Declaring classes in Python

Throughout this book, we will work with Python 3.4.3. However, all the explanations and code samples are compatible with Python 3.x.x. Therefore, you can work with previous Python versions as long as the major version number is 3. We will use JetBrains PyCharm Community Edition 4 as the main Python IDE and the supplier of an interactive Python console. However, you can use your favorite Python IDE or just the Python console.

> Everything is a class in Python, that is, all the elements that can be named in Python are classes. Guido van Rossum designed Python according to the first-class everything principle. Thus, all the types are classes, from the simplest types to the most complex ones: integers, strings, lists, dictionaries, and so on. This way, there is no difference between an integer (`int`), a `string`, and a `list`. Everything is treated in the same way. Even functions, methods, and modules are classes.

For example, when we enter the following lines in a Python console, we create a new instance of the `int` class. The console will display `<class 'int'>` as a result of the second line. This way, we know that `area` is an instance of the `int` class:

```
area = 250
type(area)
```

When we type the following lines in a Python console, we create a new instance of the `function` class. The console will display `<class 'function'>` as a result of the second line. Thus, `calculateArea` is an instance of the `function` class:

```
def calculateArea(width, height):
    return width * height

type(CalculateArea)
```

Let's analyze the simple `calculateArea` function. This function receives two arguments: width and height. It returns the width value multiplied by the height value. If we call the function with two `int` values, that is, two `int` instances, the function will return a new instance of `int` with the result of `width` multiplied by `height`. The following lines call the `calculateArea` function and save the returned `int` instance in the `rectangleArea` variable. The console will display `<class 'int'>` as a result of the third line. Thus, `rectangleArea` is an instance of the `int` class:

```
rectangleArea = calculateArea(300, 200)
print(rectangleArea)
type(rectangleArea)
```

The following lines create a new minimal `Rectangle` class in Python:

```
class Rectangle:
    pass
```

The `class` keyword followed by the class name (`Rectangle`) and a colon (`:`) composes the header of the class definition. In this case, the class doesn't have a parent class or a superclass. Therefore, there aren't superclasses enclosed in parentheses after the class name and before the colon (`:`). The indented block of statements that follows the class definition composes the body of the class. In this case, there is just a single statement, `pass`, and the class doesn't define either attributes or methods. The `Rectangle` class is the simplest possible class we can declare in Python.

[Any new class you create that doesn't specify a superclass will be a subclass of the `builtins.object` class. Thus, the `Rectangle` class is a subclass of `builtins.object`.]

The following line prints `True` as a result in a Python console, because the `Rectangle` class is a subclass of `object`:

```
issubclass(Rectangle, object)
```

The following lines represent an equivalent way of creating the `Rectangle` class in Python. However, we don't need to specify that the class inherits from an object because it adds unnecessary boilerplate code:

```
class Rectangle(object):
    pass
```

Customizing constructors in Python

We want to initialize instances of the `Rectangle` class with the values of both `width` and `height`. After we create an instance of a class, Python automatically calls the `__init__` method. Thus, we can use this method to receive both the `width` and `height` arguments. We can then use these arguments to initialize attributes with the same names. We can think of the `__init__` method as the equivalent of a constructor in other object-oriented programming languages.

The following lines create a `Rectangle` class and declare an `__init__` method within the body of the class:

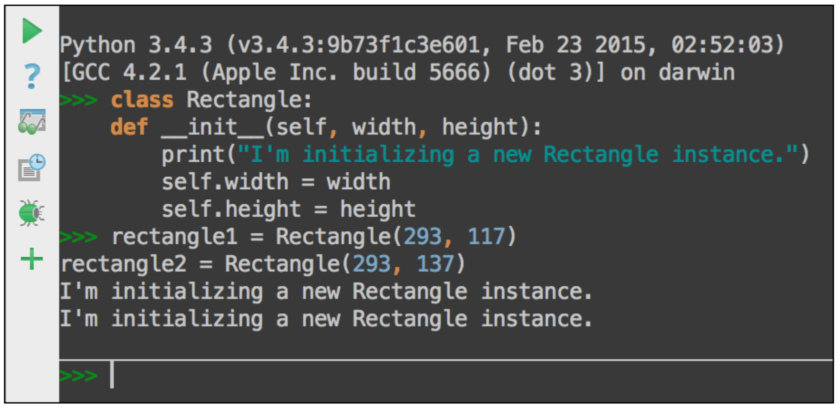
```
class Rectangle:
    def __init__(self, width, height):
        print("I'm initializing a new Rectangle instance.")
        self.width = width
        self.height = height
```

This method receives three arguments: `self`, `width`, and `height`. The first argument is a reference to the instance that called the method. We used the name `self` for this argument. It is important to notice that `self` is not a Python keyword. It is just the name for the first argument, and it is usually called `self`. The code within the method prints a message indicating that the code is initializing a new `Rectangle` instance. This way, we will understand when the code within the `__init__` method is executed.

Then, the following two lines create the `width` and `height` attributes for the instance and assign them the values received as arguments with the same names. We use `self.width` and `self.height` to create the attributes for the instance. We create two attributes for the `Rectangle` instance right after its creation.

The following lines create two instances of the `Rectangle` class named `rectangle1` and `rectangle2`. The Python console will display `I'm initializing a new Rectangle instance.` after we enter each line in the Python console:

```
rectangle1 = Rectangle(293, 117)
rectangle2 = Rectangle(293, 137)
```



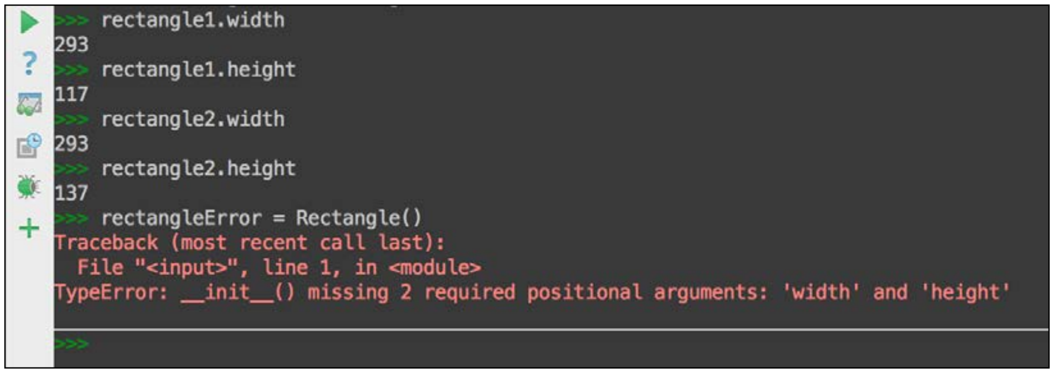
```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
>>> class Rectangle:
    def __init__(self, width, height):
        print("I'm initializing a new Rectangle instance.")
        self.width = width
        self.height = height
>>> rectangle1 = Rectangle(293, 117)
rectangle2 = Rectangle(293, 137)
I'm initializing a new Rectangle instance.
I'm initializing a new Rectangle instance.

>>> 
```

The preceding screenshot shows the Python console. Each line that creates an instance specifies the class name followed by the desired values for both the `width` and the `height` as arguments enclosed in parentheses. If we take a look at the declaration of the `__init__` method within the `Rectangle` class, we will notice that we just need to specify the second and third arguments (`width` and `height`). Also, we just need to skip the required first parameter (`self`). Python resolves many things under the hood. We just need to make sure that we specify the values for the required arguments after `self` to successfully create and initialize an instance of `Rectangle`.

After we execute the previous lines, we can check the values for `rectangle1.width`, `rectangle1.height`, `rectangle2.width`, and `rectangle2.height`.

The following line will generate a `TypeError` error and won't create an instance of `Rectangle` because we missed the two required arguments: `width` and `height`. The specific error message is `TypeError: __init__() missing 2 required positional arguments: 'width' and 'height'`. The error message is shown in the following screenshot:

```
rectangleError = Rectangle()
```



```
>>> rectangle1.width
293
>>> rectangle1.height
117
>>> rectangle2.width
293
>>> rectangle2.height
137
>>> rectangleError = Rectangle()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: __init__() missing 2 required positional arguments: 'width' and 'height'

>>>
```

Customizing destructors in Python

We want to know when the instances of the `Rectangle` class are removed from memory, that is, when the objects become inaccessible and get deleted by the garbage-collection mechanism. However, it is very important to notice that the ways in which garbage collection works depends on the implementation of Python. Remember that, Python runs on a wide variety of platforms.

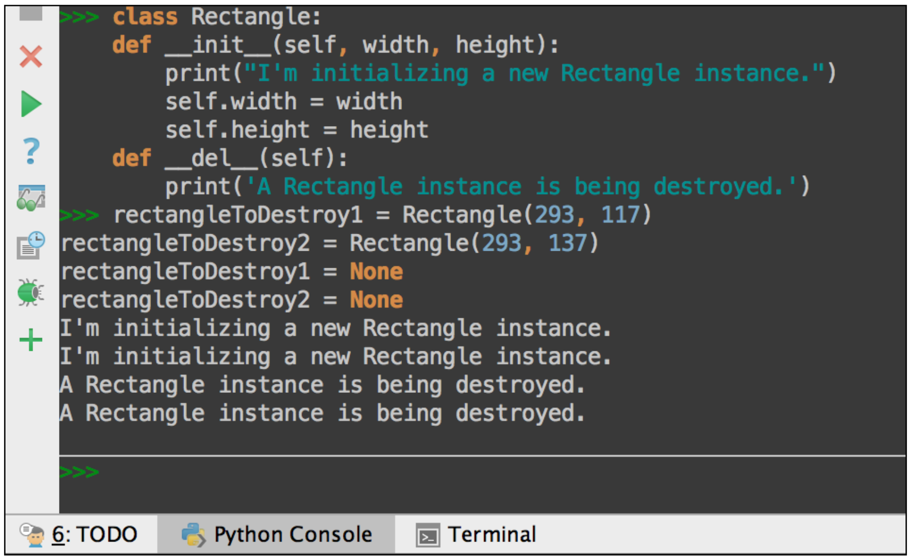
Before Python removes an instance from memory, it calls the `__del__` method. Thus, we can use this method to add any code we want to run before the instance is destroyed. We can think of the `__del__` method as the equivalent of a destructor in other object-oriented programming languages.

The following lines declare a `__del__` method within the body of the `Rectangle` class. Remember that Python always receives `self` as the first argument for any instance method:

```
def __del__(self):
        print('A Rectangle instance is being destroyed.')
```

The following lines create two instances of the `Rectangle` class: `rectangleToDelete1` and `rectangleToDelete2`. Then, the next lines assign `None` to both variables. Therefore, the reference count for both objects reaches `0`, and the garbage-collection mechanism deletes them. The Python console will display `I'm initializing a new Rectangle instance.` and then `A Rectangle instance is being destroyed.` twice in the Python console. Python executes the code within the `__del__` method after we assign `None` to each variable that had the only reference to an instance of the `Rectangle` class:

```
rectangleToDestroy1 = Rectangle(293, 117)
rectangleToDestroy2 = Rectangle(293, 137)
rectangleToDestroy1 = None
rectangleToDestroy2 = None
```



```
>>> class Rectangle:
    def __init__(self, width, height):
        print("I'm initializing a new Rectangle instance.")
        self.width = width
        self.height = height
    def __del__(self):
        print('A Rectangle instance is being destroyed.')
>>> rectangleToDestroy1 = Rectangle(293, 117)
rectangleToDestroy2 = Rectangle(293, 137)
rectangleToDestroy1 = None
rectangleToDestroy2 = None
I'm initializing a new Rectangle instance.
I'm initializing a new Rectangle instance.
A Rectangle instance is being destroyed.
A Rectangle instance is being destroyed.

>>>
```

You can add some cleanup code within the `__del__` method. However, take into account that most of the time, you can follow best practices to release resources without having to add code to the `__del__` method. Remember that you don't know exactly when the `__del__` method is going to be executed. Even when the reference count reaches `0`, the Python implementation might keep the resources until the appropriate garbage collection destroys the instances.

The following lines create a `rectangle3` instance of the `Rectangle` class and then assign a `referenceToRectangle3` reference to this object. Thus, the reference count to the object increases to `2`. The next line assigns `None` to `rectangle3`, and therefore, the reference count for the object goes down from `2` to `1`. As the `referenceToRectangle3` variable stills holds a reference to the `Rectangle` instance, Python doesn't destroy the instance, and we don't see the results of the execution of the `__del__` method:

```
rectangle3 = Rectangle(364, 257)
referenceToRectangle3 = rectangle3
rectangle3 = None
```

Python destroys the instance if we add a line that assigns `None` to `referenceToRectangle3`:

```
referenceToRectangle3 = None
```

However, it is very important to know that you don't need to assign `None` to a reference to force Python to destroy objects. In the previous examples, we wanted to understand how the `__del__` method worked. Python will automatically destroy the objects when they aren't referenced anymore.

Creating instances of classes in Python

We already created instances of the simple `Rectangle` class. We just needed to use the class name, specify the required arguments enclosed in parentheses, and assign the result to a variable.

The following lines declare a `calculate_area` method within the body of the `Rectangle` class:

```
def calculate_area(self):
    return self.width * self.height
```

The method doesn't require arguments to be called from an instance because it just declares the previously explained `self` parameter. The following lines create an instance of the `Rectangle` class named `rectangle4` and then print the results of the call to the `calculate_area` method for this object:

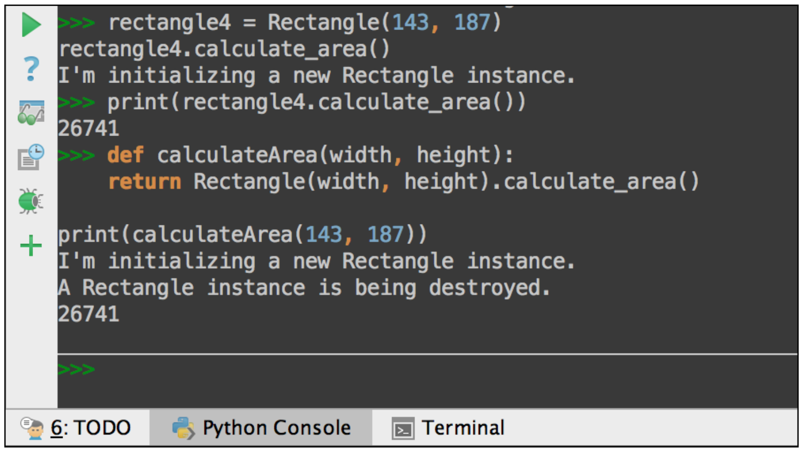
```
rectangle4 = Rectangle(143, 187)
print(rectangle4.calculate_area())
```

Now, imagine that we want to have a function that receives the width and height values of a rectangle and returns the calculated area. We can take advantage of the `Rectangle` class to code this new function. We just need to create an instance of the `Rectangle` class with the `width` and `height` received as parameters and return the result of the call to the `calculate_area` method. Remember that we don't have to worry about releasing the resources required by the `Rectangle` instance, because the reference count for this object will become `0` after the function returns the result. The following lines show the code for the `calculateArea` independent function, which isn't part of the `Rectangle` class body:

```
def calculateArea(width, height):
    return Rectangle(width, height).calculate_area()

print(calculateArea(143, 187))
```

Notice that the Python console displays the following messages. Thus, we can see that the instance is destroyed and the code within the `__del__` method is executed. The messages are shown in the following screenshot:

```
I'm initializing a new Rectangle instance.
A Rectangle instance is being destroyed.
26741
```



```
>>> rectangle4 = Rectangle(143, 187)
rectangle4.calculate_area()
I'm initializing a new Rectangle instance.
>>> print(rectangle4.calculate_area())
26741
>>> def calculateArea(width, height):
    return Rectangle(width, height).calculate_area()

print(calculateArea(143, 187))
I'm initializing a new Rectangle instance.
A Rectangle instance is being destroyed.
26741
>>>
```

Declaring classes in C#

Throughout this book, we will work with C# 6.0 (introduced in Microsoft Visual Studio 2015). However, most of the explanations and code samples are also compatible with C# 5.0 (introduced in Visual Studio 2013). If a specific example uses C# 6.0 syntax and isn't compatible with C# 5.0, the code will be properly labeled with the compatibility warning. We will use Visual Studio Community 2015 as the main IDE. However, you can also run the examples using Mono or Xamarin.

The following lines declare a new minimal `Circle` class in C#:

```
class Circle
{
}
```

The `class` keyword followed by the class name (`Circle`) composes the header of the class definition. In this case, the class doesn't have a parent class or a superclass. Therefore, there aren't any superclasses listed after the class name and a colon (`:`). A pair of curly braces (`{}`) encloses the class body after the class header. In this case, the class body is empty. The `Circle` class is the simplest possible class we can declare in C#.

> Any new class you create that doesn't specify a superclass will be a subclass of the `System.Object` class in C#. Thus, the `Circle` class is a subclass of `System.Object`.

The following lines represent an equivalent way of creating the `Circle` class in C#. However, we don't need to specify that the class inherits from `System.Object`, because it adds unnecessary boilerplate code:

```
class Circle: System.Object
{
}
```

Customizing constructors in C#

We want to initialize instances of the `Circle` class with the radius value. In order to do so, we can take advantage of the constructors in C#. Constructors are special class methods that are automatically executed when we create an instance of a given type. The runtime executes the code within the constructor before any other code within a class.

We can define a constructor that receives the radius value as an argument and use it to initialize an attribute with the same name. We can define as many constructors as we want. Therefore, we can provide many different ways of initializing a class. In this case, we just need one constructor.

The following lines create a `Circle` class and define a constructor within the class body.

```
class Circle
{
  private double radius;

  public Circle(double radius)
  {
    Console.WriteLine(String.Format("I'm initializing a new Circle
instance with a radius value of {0}.", radius));
    this.radius = radius;
  }
}
```

The constructor is a public class method that uses the same name as the class: `Circle`. The method receives a single argument: `radius`. The code within the method prints a message on the console, indicating that the code is initializing a new `Circle` instance with a specific radius value. This way, we will understand when the code within the constructor is executed. As the constructor has an argument, it is known as a parameterized constructor.

Then, the following line assigns the radius double value received as an argument to the private radius double field. We use `this.radius` to access the private radius attribute for the instance and `radius` to reference the argument. In C#, the `this` keyword provides access to the instance that has been created and the one we want to initialize. The line before the constructor declares the private `radius` double field. At this time, we won't pay attention to the difference between the `private` and `public` keywords. We will dive deep into the proper usage of these keywords in *Chapter 3, Encapsulation of Data*.

The following lines create two instances of the `Circle` class: `circle1` and `circle2`. The **Windows Console** application will display `I'm initializing a new Circle instance with a radius value of`, followed by the radius value specified in the call to the constructor of each instance:

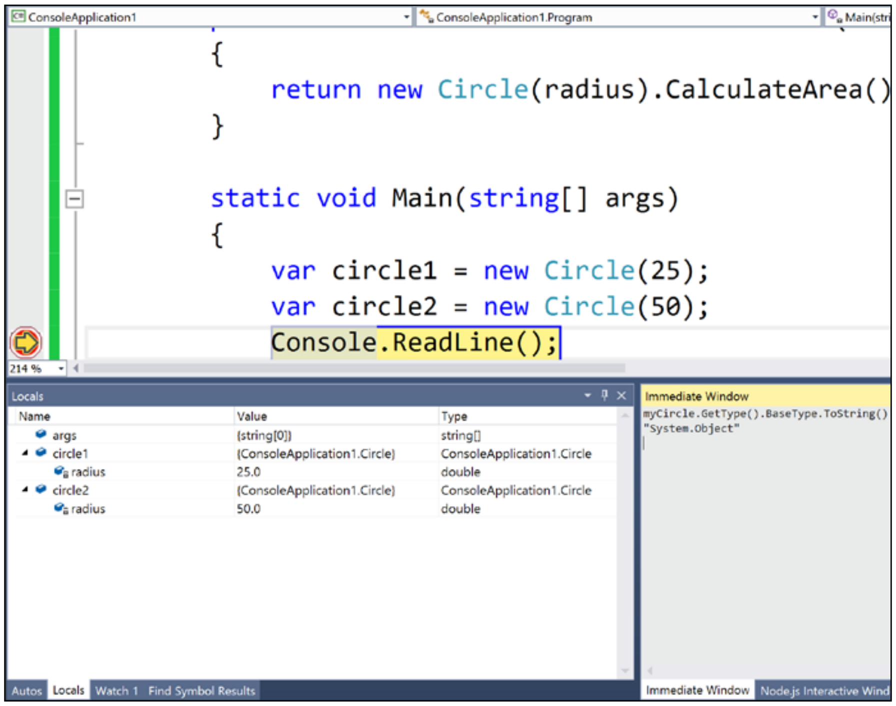
```
class Chapter01
{
  public static void Main(string[] args)
  {
```

```
            var circle1 = new Circle(25);
            var circle2 = new Circle(50);
            Console.ReadLine();
        }
    }
```

Each line that creates an instance uses the `new` keyword, followed by the desired value for the radius as an argument enclosed in parentheses. We used the `var` keyword to let C# automatically assign the `Circle` type for each of the variables. After we execute the two lines that create the instances of `Circle`, we can use an inspector, such as the **Autos Window**, the **Watch Window**, or the **Immediate Window**, to check the values for `circle1.radius` and `circle2.radius`.

The following line prints **"System.Object"** as a result in the **Immediate Window** in the IDE. This is because the `Circle` class is a subclass of `System.Object`:

```
    circle1.GetType().BaseType.ToString()
```

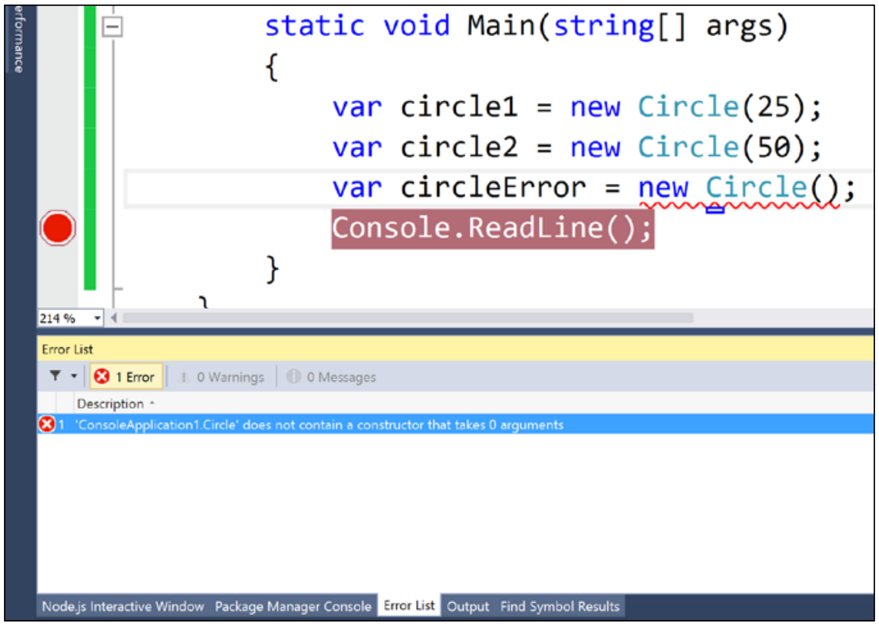


```
        {
            return new Circle(radius).CalculateArea()
        }

        static void Main(string[] args)
        {
            var circle1 = new Circle(25);
            var circle2 = new Circle(50);
            Console.ReadLine();
```

Name	Value	Type
args	{string[0]}	string[]
circle1	{ConsoleApplication1.Circle}	ConsoleApplication1.Circle
radius	25.0	double
circle2	{ConsoleApplication1.Circle}	ConsoleApplication1.Circle
radius	50.0	double

```
myCircle.GetType().BaseType.ToString()
"System.Object"
```

The following line won't allow the console application to compile and will display a build error. This is because the compiler cannot find a parameterless constructor declared in the `Circle` class. The specific error message is `ConsoleApplication does not contain a constructor that takes 0 arguments`. The following screenshot displays the `var circleError = new Circle();` error:



```
static void Main(string[] args)
{
    var circle1 = new Circle(25);
    var circle2 = new Circle(50);
    var circleError = new Circle();
    Console.ReadLine();
}
```

```
Error List
1 Error    0 Warnings    0 Messages
Description
1  'ConsoleApplication1.Circle' does not contain a constructor that takes 0 arguments
```

Customizing destructors in C#

We want to know when the instances of the `Circle` class are removed from memory, that is, when the objects go out of scope and the garbage-collection mechanism removes them from memory. Destructors are the special class methods that are automatically executed when the run time destroys an instance of a given type. Thus, we can use them to add any code we want to run before the instance is destroyed.

The destructor is a special class method that uses the same name as the class, but prefixed with a tilde (~): `~Circle`. The destructor must be parameterless, and it cannot return a value.

The following lines declare a destructor (a `~Circle` method) within the body of the `Circle` class:

```
~Circle()
{
  Console.WriteLine(String.Format("I'm destroying the Circle instance
with a radius value of {0}.", radius));
}
```

The code within the destructor prints a message on the console indicating that the runtime is destroying a `Circle` instance with a specific radius value. This way, we will understand when the code within the destructor is executed.

If we execute the console application after adding the code for the destructor to the `Circle` class, we will see the following lines in the console output. The first two lines will appear before we press a key. After we press a key, we will see the two lines indicating that the code within the destructor has been executed. This is because the two variables `circle1` and `circle2` have run out of scope and the garbage collector has destroyed the objects:

```
I'm initializing a new Circle instance with a radius value of 25.
I'm initializing a new Circle instance with a radius value of 50.

I'm destroying the Circle instance with a radius value of 50.
I'm destroying the Circle instance with a radius value of 25.
```

Creating instances of classes in C#

We already created instances of the simple `Circle` class. We just needed to use the `new` keyword followed by the class name, specify the required arguments enclosed in parentheses, and assign the result to a variable.

The following lines declare a public `CalculateArea` method within the body of the `Circle` class:

```
public double CalculateArea()
{
  return Math.PI * Math.Pow(this.radius, 2);
}
```

The method doesn't require arguments to be called. It returns the result of the multiplication of π by the square of the radius field value (`this.radius`). The following lines show a new version of the `Main` method. These lines create two instances of the `Circle` class: `circle1` and `circle2`. The lines then display the results of calling the `CalculateArea` method for the two objects. The new lines are highlighted, as follows:

```
class Chapter01
{
  public static void Main(string[] args)
  {
    var circle1 = new Circle(25f);
    var circle2 = new Circle(50f);
    Console.WriteLine(String.Format("The area for circle #1 is {0}",
circle1.CalculateArea()));
    Console.WriteLine(String.Format("The area for circle #2 is {0}",
circle2.CalculateArea()));
    Console.ReadLine();
  }
}
```

Now, imagine that we want to have a function that receives the radius value of a circle and has to return the calculated area. We can take advantage of the `Circle` class to code this new function. We just need to create an instance of the `Circle` class with the radius received as a parameter and return the result of the call to the `CalculateArea` method. Remember that, we don't have to worry about releasing the resources required by the `Circle` instance, because the object will go out of scope after the function returns the result. The following lines show the code for the new `CalculateCircleArea` function that isn't part of the `Circle` class body. The function is a method of the *Chapter 1, Objects Everywhere* class body, which also has the `Main` method:

```
class Chapter01
{
  private static double CalculateCircleArea(double radius)
  {
    return new Circle(radius).CalculateArea();
  }

  static void Main(string[] args)
  {
    double radius = 50;
    Console.WriteLine(String.Format("The area for a circle with a
radius of {0} is {1} ", radius, CalculateCircleArea(radius)));
    Console.ReadLine();
  }
}
```

The Windows command line displays the following messages. Thus, we can see that the instance is destroyed and the code within the destructor is executed:

```
I'm initializing a new Circle instance with a radius value of 50.
The area for a circle with a radius of 50 is 7853.98163397448
I'm destroying the Circle instance with a radius value of 50.
```

Understanding that functions are objects in JavaScript

We will use **Chrome Developer Tools** (**CDT**), as the main JavaScript console. However, you can run the examples in any other browser that provides a JavaScript console.

Functions are first-class citizens in JavaScript. In fact, functions are objects in JavaScript. When we type the following lines in a JavaScript console, we create a new function object. Thus, `calculateArea` is an object, and its type is `function`. Notice the results of writing the following lines in a JavaScript console. The displayed type for `calculateArea` is a function, as follows:

```
function calculateArea(width, height) { return width * height; }
typeof(calculateArea)
```

The `calculateArea` function receives two arguments: `width` and `height`. It returns the width value multiplied by the height value. The following line calls the `calculateArea` function and saves the returned number in the `rectangleArea` variable:

```
var rectangleArea = calculateArea(300, 200);
console.log(rectangleArea);
```

Functions are special objects in JavaScript that contain code and that you can invoke. They contain properties and methods. For example, if we type the following line, the JavaScript console will display the value for the `name` property of the function object, that is, the `calculateArea` function:

```
console.log(calculateArea.name);
```

Downloading the example code

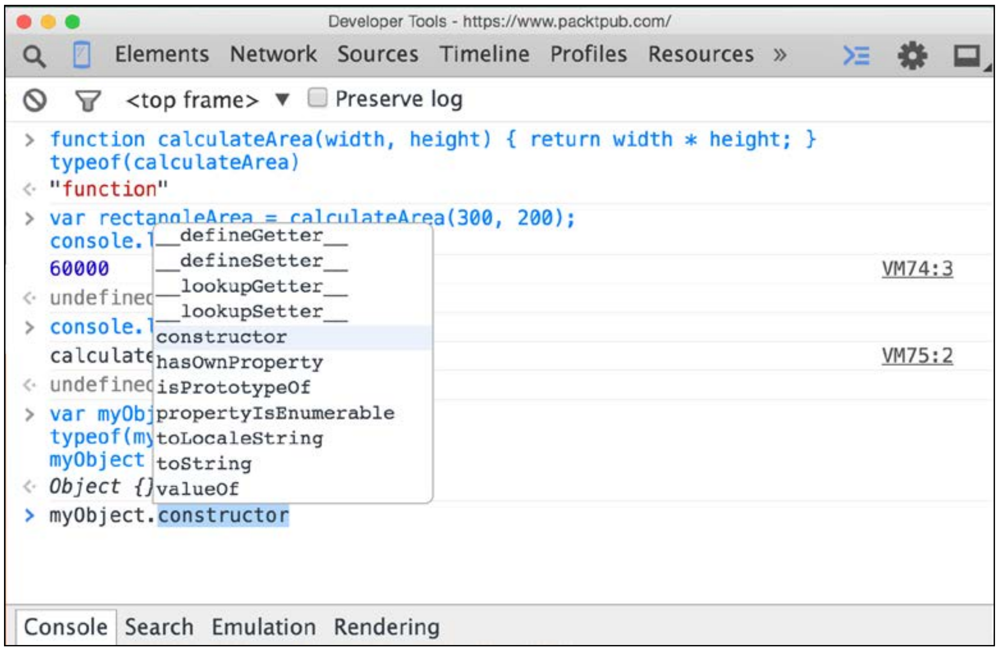
You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Working with constructor functions in JavaScript

The following lines of code create an object named `myObject` without any specific properties or methods. This line checks the type of the variable (`myObject`) and then prints the key-value pairs that define the object on the JavaScript console:

```
var myObject = {};
typeof(myObject);
myObject
```

The preceding lines created an empty object. Therefore, the result of the last line shows `Object {}` on the console. There are no properties or methods defined in the object. However, if we enter `myObject.` (`myObject` followed by a dot) in a JavaScript console with autocomplete features, we will see many properties and methods listed, as shown in the following screenshot. The object includes many built-in properties and methods:


Developer Tools - https://www.packtpub.com/
Elements Network Sources Timeline Profiles Resources »
<top frame> ▼ Preserve log
Console Search Emulation Rendering

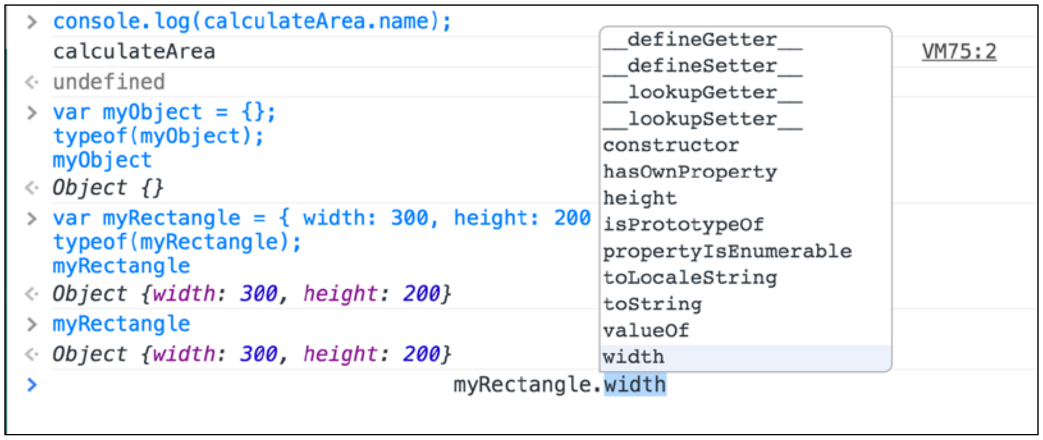
```
> function calculateArea(width, height) { return width * height; }
  typeof(calculateArea)
<· "function"
> var rectangleArea = calculateArea(300, 200);
  console.l
  60000                                                    VM74:3
<· undefined
> console.l
  calculate                                                VM75:2
<· undefined
> var myObj
  typeof(my
  myObject
<· Object {}
> myObject.constructor
```

```
__defineGetter__
__defineSetter__
__lookupGetter__
__lookupSetter__
constructor
hasOwnProperty
isPrototypeOf
propertyIsEnumerable
toLocaleString
toString
valueOf
```

The following lines of code create an object named `myRectangle` with two key-value pairs enclosed within a pair of curly braces (`{}`). A colon (`:`) separates the key from the value and a comma (`,`) separates the key-value pairs. The next line checks the type of the variable (`object`) and prints the key-value pairs that define the object on the JavaScript console:

```
var myRectangle = { width: 300, height: 200 };
typeof(myRectangle);
myRectangle
```

The preceding lines created an object with two properties: `width` and `height`. The result of the last line shows `Object {width: 300, height: 200}` on the console. Thus, the `width` property has an initial value of `300`, and the `height` property has an initial value of `200`. If we enter `myRectangle.` (`myRectangle` followed by a dot) in a JavaScript console with autocomplete features, we will see the `width` and `height` properties listed with the built-in properties and methods, as shown in the following screenshot:



```
> console.log(calculateArea.name);
  calculateArea                                   VM75:2
<· undefined
> var myObject = {};
  typeof(myObject);
  myObject
<· Object {}
> var myRectangle = { width: 300, height: 200
  typeof(myRectangle);
  myRectangle
<· Object {width: 300, height: 200}
> myRectangle
<· Object {width: 300, height: 200}
>                   myRectangle.width
```

So far, we have been creating independent objects. The first example was an empty object, and the second case was an object with two properties with their initial values initialized. However, if we want to create another object with the same properties but different values, we would have to specify the property names again. For example, the following line creates another object named `myRectangle2`, with the same two keys, but different values:

```
var myRectangle2 = { width: 500, height: 150 };
```

However, we are repeating code and can have typos when we enter the key names, that is, the names for the future properties of the instance. Imagine that we had written the following line instead of the preceding line (notice that the code contains typos):

```
var myRectangle2 = { widh: 500, hight: 150 };
```

The previous line will generate `widh` and `hight` properties instead of `width` and `height`. Thus, if we want to retrieve the value from `myRectangle2.width`, we would receive `undefined` as a response. This is because `myRectangle2` created the `widh` property instead of `width`.

We want to initialize new rectangle objects with the values of both the width and the height. However, we don't want a typo to generate problems in our code. Thus, we need a blueprint that generates and initializes properties with the same names. In addition, we want to log a message to the console whenever we have a new rectangle object. In order to do so, we can take advantage of the constructor function. The following lines declare a `Rectangle` constructor function in JavaScript:

```
function Rectangle(width, height) {
  console.log("I'm creating a new Rectangle");
  this.width = width;
  this.height = height;
}
```

Notice the capitalized name of the function, `Rectangle` instead of `rectangle`. It is a good practice to capitalize constructor functions to distinguish them from the other functions.

The constructor function receives two arguments: `width` and `height`. The code within the function is able to access the new instance of the current object that has been created with the `this` keyword. Thus, you can translate `this` to the current object. The code within the function prints a message on the JavaScript console, indicating that it is creating a new rectangle. The code then uses the received `width` and `height` arguments to initialize properties with the same names. We use `this.width` and `this.height` to create the properties for the instance. We create two properties for the instance right after its creation. We can think of the constructor function as the equivalent of a constructor in other object-oriented programming languages.

The following lines create two `Rectangle` objects named `rectangle1` and `rectangle2`. Notice the usage of the `new` keyword to call the constructor function, with the `width` and `height` values enclosed in parentheses. The Python console will display `I'm initializing a new Rectangle instance.` after we enter each line in the Python console:

```
var rectangle1 = new Rectangle(293, 117);
var rectangle2 = new Rectangle(293, 137);
```

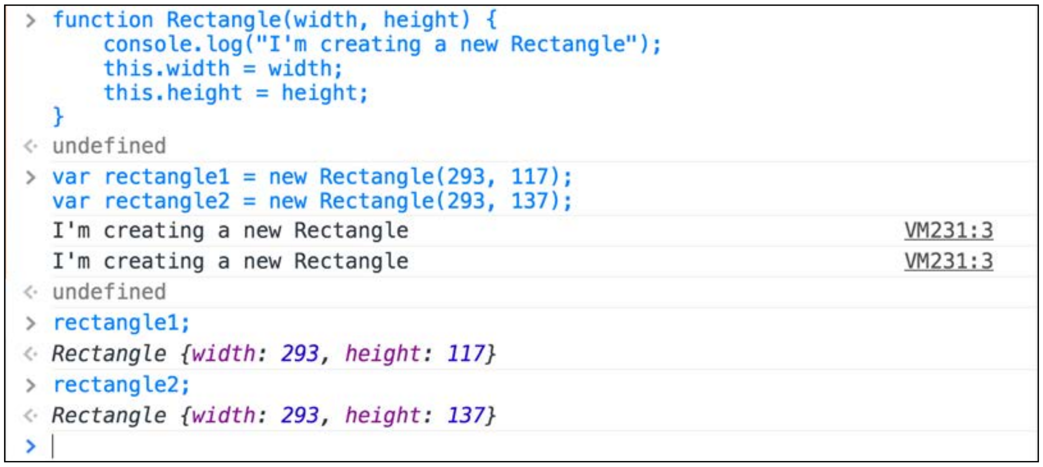
Each line that creates an instance uses the `new` keyword followed by the constructor function and the desired values for both the width and the height as arguments enclosed in parentheses. After we execute the previous lines, we can check the values for `rectangle1.width`, `rectangle1.height`, `rectangle2.width`, and `rectangle2.height`.

Enter the following two lines in the console:

```
rectangle1;
rectangle2;
```

The console will display the following two lines:

```
Rectangle {width: 293, height: 117}
Rectangle {width: 293, height: 137}
```



```
> function Rectangle(width, height) {
      console.log("I'm creating a new Rectangle");
      this.width = width;
      this.height = height;
  }
<· undefined
> var rectangle1 = new Rectangle(293, 117);
  var rectangle2 = new Rectangle(293, 137);
  I'm creating a new Rectangle                              VM231:3
  I'm creating a new Rectangle                              VM231:3
<· undefined
> rectangle1;
<· Rectangle {width: 293, height: 117}
> rectangle2;
<· Rectangle {width: 293, height: 137}
> |
```

It is very clear that we have created two `Rectangle` objects and not just two `Object` objects. We can see that the constructor function name appears before the key-value pairs.

Enter the following line in the console:

```
rectangle1 instanceof Rectangle
```

The console will display `true` as a result of the evaluation of the previous expression, because `rectangle1` is an instance of `Rectangle`. This way, we can determine whether an object is a `Rectangle` object, that is, an instance created using the `Rectangle` constructor function.

Creating instances in JavaScript

We already created instances with the simple `Rectangle` constructor function. We just needed to use the `new` keyword and the constructor function name. We then need to specify the required arguments enclosed in parentheses and assign the result to a variable.

The following lines declare a new version of the `Rectangle` constructor function that adds a `calculateArea` function to the blueprint:

```
function Rectangle(width, height) {
    console.log("I'm creating a new Rectangle");
    this.width = width;
    this.height = height;

    this.calculateArea = function() {
        return this.width * this.height;
    }
}
```

The new constructor function adds a parameterless `calculateArea` method to the instance. The following lines of code create a new `Rectangle` object named `rectangle3` and then print the results of the call to the `calculateArea` method for this object:

```
var rectangle3 = new Rectangle(143, 187);
rectangle3.calculateArea();
```

If we enter the following line, the JavaScript console will display the same code we entered to create the new version of the `Rectangle` constructor function. Thus, we might create a new `Rectangle` object by calling the `rectangle3.constructor` function in the next line. Remember that the `constructor` property is automatically generated by JavaScript, and this property in is a function:

```
var rectangle4 = new rectangle3.constructor(300, 200);
```

Now, imagine that we want to have a function that receives the width and height values of a rectangle and returns the calculated area. We can take advantage of a `Rectangle` object to code this new function. We just need to create a `Rectangle` object using the `Rectangle` constructor function with `width` and `height` received as parameters. We then need to return the result of the call to the `calculateArea` method. Remember that we don't have to worry about releasing the resources required by the `Rectangle` object, because the variable will go out of scope after the function returns the result. The following lines show the code for the `calculateArea` independent function, which isn't a part of the `Rectangle` constructor function:

```
function calculateArea(width, height) {
  return new Rectangle(width, height).calculateArea();
}

calculateArea(143, 187);
```

Summary

In this chapter, you learned about an object's life cycle. You also learned how object constructors and destructors work. We declared classes in Python and C# and used constructor functions in JavaScript to generate blueprints for objects. We customized constructors and destructors, and tested their personalized behavior in action. We understood different ways of generating instances in the three programming languages.

Now that you have learned to start creating classes and instances, we are ready to share, protect, and hide data with the data-encapsulation features of Python, JavaScript, and C#, which is the topic of the next chapter.

3
Encapsulation of Data

In this chapter, we will start organizing data in blueprints that generate objects. We will protect and hide data in each of the three covered programming languages: Python, JavaScript, and C#. We will:

- Understand the different members of a class
- Learn the difference between mutability and immutability
- Customize methods and fields to protect them against undesired access
- Work with access modifiers, naming conventions, and properties
- Customize getter and setter methods
- Create properties with getters and setters in Python, C#, and JavaScript

Understanding the different members of a class

So far, we have been working with simple classes and used them to generate instances in Python and C#. We also defined the functions of a constructor to generate objects based on prototypes in JavaScript. Now, it's time to dive deeper to understand the different members of a class.

The following list enumerates all the possible element types that you can include in a class definition. We already worked with many of these elements:

- Constructor
- Destructor
- Class fields or class attributes
- Class methods
- Nested classes

- Instance fields or instance attributes
- Properties with getters and/or setters
- Events

You already learned how constructors and destructors work in Python and C#. We also worked with constructor functions in JavaScript. So far, we have been using instance fields, also known as instance attributes, to encapsulate data in our instances. We can access these fields or attributes without any kind of restriction as variables within an instance. We also worked with instance methods that we could invoke without any kind of restrictions.

However, as happens in real-world situations, sometimes restrictions are necessary to avoid serious problems. Sometimes, we want to restrict access or transform-specific instance fields in read-only attributes. The different programming languages take different approaches that allow you to establish restrictions for the different members of a class. We can combine these restrictions with properties that can define getters and/or setters.

Properties can define get and/or set methods, also known as **getters** and **setters**. Setters allow you to control how values are set to protected instance attributes, that is, these methods are used to change the values of underlying instance attributes. Getters allow you to control how values are returned. Getters don't change the values of the underlying attributes.

Sometimes, all the members of a class share the same field or attribute, and we don't need to have a specific value for each instance. For example, the dog breeds have some profile values, such as the average size of males (width and height) and the average size of females (width and height). We can define the following class fields to store the values that are shared by the `averageMaleWidth`, `averageMaleHeight`, `averageFemaleWidth`, and `averageFemaleHeight` instances. All these instances have access to the same class field and their values. However, it's also possible to apply restrictions to their access.

Events allow instances to notify other objects when an event takes place. A publisher instance raises or sends an event, whereas a subscriber instance receives or handles the event. Instances can subscribe to events to add the necessary code to be executed when an event is raised, that is, when something of interest occurs. You can think about events as a mechanism to generate subscriber-publisher relationships between instances. For example, you can use events to make a dog bark when another dog arrives near it.

It's also possible to define methods that don't require an instance of a specific class to be called; therefore, you can invoke them by specifying the class name and the method name. These methods are known as class methods; they operate on a class as a whole and have access to class fields, but they don't have access to any instance members (such as instance fields, properties, or methods) because there is no instance at all. The class methods are useful when you want to include methods related to a class; you don't want to generate an instance to call them.

Protecting and hiding data

When we design classes, we want to make sure that all the necessary data is available to the methods that will operate on this data; therefore, we encapsulate the data. However, we just want the relevant information to be visible to the users of our classes that will create instances, change values of accessible attributes or properties, and call the available methods. Therefore, we want to hide or protect some data that is just needed for internal use. We don't want accidental changes to sensitive data.

For example, when we create a new instance of any dog breed, we can use its name and birth date as two parameters for a constructor. The constructor initializes the values of two internal fields: `m_name` and `m_birthDate`.

We don't want a user of our dog breed class to be able to change a dog's name after an instance has been initialized because the name is not supposed to change. Thus, we define a property called `Name` with a getter method, but without a setter method. This way, it's possible to retrieve the dog's name, but we cannot change it because there isn't a setter method defined for the property. The getter method just returns the value of the `m_name` internal field. In addition, we can protect the internal `m_name` field to avoid the users of a class to access the `m_name` field from any instance of the class.

We don't want a user of our dog breed class to be able to change a dog's birth date after an instance has been initialized because the dog won't be born again on a different date. In fact, we just want to make the dog's age available to different users. Thus, we define a property called `Age` with a getter method, but without a setter method. This way, it's possible to retrieve the dog's age, but we cannot change it because there isn't a setter method defined for a property. The getter method returns the result of calculating the dog's age based on the current date and the value of the `m_birthDate` internal field. As in the previous example, we can also protect the internal `m_birthDate` field to prevent the users of a class to access a field from any instance of a class.

This way, our class can make two read-only properties, `Name` and `Age`, public. The `m_name` and `m_birthDate` internal fields aren't public and cannot be accessed from the instances of a class. The read-only properties expose the values of all internal fields.

Working with properties

It's possible to manually add getter and setter methods to emulate how properties work. For example, we can add a `GetName` method that just returns the value of the `m_name` internal field. Each time we want to retrieve a dog's name, it will be necessary to call the `GetName` method for that specific instance.

A dog's favorite toy may change over time. However, we still want to use getter and setter methods to keep control over the procedure of retrieving and setting the value of an underlying `m_favoriteToy` internal field. We always want users to change the values of a field using the getter and setter methods, just in case we need to add some code within these methods in the future. For example, we can decide that whenever a dog's favorite toy changes, it's necessary to update the dog's playfulness score. If we force the user to use the setter method whenever he/she needs to update the dog's favorite toy, we can easily add the necessary code that updates the dog's playfulness score within this setter method.

We can manually add both getter and setter methods to emulate how properties work for the `m_favoriteToy` internal field. We have to add a `GetFavoriteToy` getter method and a `SetFavoriteToy` setter method. This way, whenever we want to retrieve a dog's favorite toy, it will be necessary to call the `GetFavoriteToy` method for a specific instance. Whenever we want to specify a new value to a dog's favorite toy, it will be necessary to call the `SetFavoriteToy` method with the new favorite toy as an argument.

The getter and setter methods combined with the access protection of the `m_favoriteToy` internal field make it possible to have absolute control over the way in which the dog's favorite toy is retrieved and set. However, it would be nicer to define a public property named `FavoriteToy`. Whenever we assign a value to the `FavoriteToy` property, the `SetFavoriteToy` setter method is called under the hood with the value to be assigned as an argument. Whenever we specify the `FavoriteToy` property in any expression, the `GetFavoriteToy` getter method is called under the hood to retrieve the actual value.

[Each programming language provides a different mechanism and syntax to define properties and customize the getter and setter methods. Based on our goals, we can combine properties, getter and setters methods, the underlying fields, and the access protection mechanisms.]

Understanding the difference between mutability and immutability

By default, any instance field or attribute works like a variable; therefore, we can change their values. When we create an instance of a class that defines many public instance fields, we are creating a mutable object, that is, an object that can change its state.

For example, let's think about a class named `MutableVector3D` that represents a mutable 3D vector with three public instance fields: `X`, `Y`, and `Z`. We can create a new `MutableVector3D` instance and initialize the `X`, `Y`, and `Z` attributes. Then, we can call the `Sum` method with their delta values for `X`, `Y`, and `Z` as arguments. The delta values specify the difference between the existing value and the new or desired value. So, for example, if we specify a positive value of `20` in the `deltaX` parameter, it means that we want to add `20` to the `X` value.

The following lines show pseudocode in a neutral programming language that create a new `MutableVector3D` instance called `myMutableVector`, initialized with values for the `X`, `Y`, and `Z` fields. Then, the code calls the `Sum` method with the delta values for `X`, `Y`, and `Z` as arguments, as shown in the following code:

```
myMutableVector = new MutableVector3D instance with X = 30, Y = 50 and
Z = 70
myMutableVector.Sum(deltaX: 20, deltaY: 30, deltaZ: 15)
```

The initial values for the `myMutableVector` field are `30` for `X`, `50` for `Y`, and `70` for `Z`. The `Sum` method changes the values of all the three fields; therefore, the object state mutates as follows:

- `myMutableVector.X` mutates from `30` to `30 + 20 = 50`
- `myMutableVector.Y` mutates from `50` to `50 + 30 = 80`
- `myMutableVector.Z` mutates from `70` to `70 + 15 = 85`

> The values for the `myMutableVector` field after the call to the `Sum` method are `50` for `X`, `80` for `Y`, and `85` for `Z`. We can say this method mutated the object's state; therefore, `myMutableVector` is a mutable object: an instance of a mutable class.

Mutability is very important in object-oriented programming. In fact, whenever we expose fields and/or properties, we will create a class that will generate mutable instances. However, sometimes a mutable object can become a problem. In certain situations, we want to avoid objects to change their state. For example, when we work with a concurrent code, an object that cannot change its state solves many concurrency problems and avoids potential bugs.

For example, we can create an immutable version of the previous `MutableVector3D` class to represent an immutable 3D vector. The new `ImmutableVector3D` class has three read-only properties: `X`, `Y`, and `Z`. Thus, there are only three getter methods without setter methods, and we cannot change the values of the underlying internal fields: `m_X, m_Y,` and `m_Z`. We can create a new `ImmutableVector3D` instance and initialize the underlying internal fields: `m_X, m_Y,` and `m_Z`. `X`, `Y`, and `Z` attributes. Then, we can call the `Sum` method with the delta values for `X`, `Y`, and `Z` as arguments.

The following lines show the pseudocode in a neutral programming language that create a new `ImmutableVector3D` instance called `myImmutableVector`, which is initialized with values for `X`, `Y`, and `Z` as arguments. Then, the pseudocode calls the `Sum` method with the delta values for `X`, `Y`, and `Z` as arguments:

```
myImmutableVector = new ImmutableVector3D instance with X = 30, Y = 50
and Z = 70
myImmutableSumVector = myImmutableVector.Sum(deltaX: 20, deltaY: 30,
deltaZ: 15)
```

However, this time the `Sum` method returns a new instance of the `ImmutableVector3D` class with the `X`, `Y`, and `Z` values initialized to the sum of `X`, `Y`, and `Z` and the delta values for `X`, `Y`, and `Z`. So, `myImmutableSumVector` is a new `ImmutableVector3D` instance initialized with `X` = `50`, `Y` = `80`, and `Z` = `85`. The call to the `Sum` method generated a new instance and didn't mutate the existing object.

The immutable version adds an overhead as compared with the mutable version because it's necessary to create a new instance of a class as a result of calling the `Sum` method. The mutable version just changed the values for the attributes and it wasn't necessary to generate a new instance. Obviously, the immutable version has a memory and a performance overhead. However, when we work with the concurrent code, it makes sense to pay for the extra overhead to avoid potential issues caused by mutable objects.

Encapsulating data in Python

First, we will add attributes to a class in Python and then use prefixes to hide specific members of a class. We will use property getters and setters to control how we write and retrieve values to and from related attributes. We will use methods to add behaviors to classes, and we will create the mutable and immutable version of a 3D vector to understand the difference between an object that mutates state and an object that doesn't.

Adding attributes to a class

The `TibetanSpaniel` class is a blueprint for dogs that belong to the Tibetan Spaniel breed. This class should inherit from a `Dog` superclass, but we will forget about inheritance and other dog breeds for a while. We will use the `TibetanSpaniel` class to understand the difference between class attributes and instance attributes.

As happens with any other dog breed, Tibetan Spaniel dogs have some profile values. We will define the following class attributes to store the values that are shared by all the members of the Tibetan Spaniel breed. Note that the valid values for scores are from `0` to `10`; `0` is the lowest skill and `10` the highest.

- `family`: This is the family to which the dog breed belongs
- `area_of_origin`: This is the are a of origin of the dog breed
- `learning_rate`: This is the typical learning rate score for the members of this dog breed
- `obedience`: This is the average obedience score for the members of this dog breed
- `problem_solving`: This is the average problem solving score for the members of this dog breed

The following lines create a `TibetanSpaniel` class and declare the previously enumerated class attributes and a `__init__` method within the body of a class:

```
class TibetanSpaniel:
  family = "Companion, herding"
  area_of_origin = "Tibet"
  learning_rate = 9
  obedience = 3
  problem_solving = 8

  def __init__(self, name, favorite_toy, watchdog_ability):
    self.name = name
    self.watchdog_ability = watchdog_ability
    self.favorite_toy = favorite_toy
```

The preceding code assigns a value to each class variable after the class header within the class body and without `self.` as its prefix. This code assigns a value outside of any method because there is no need to create any instance to access the attributes of a class.

It's common practice to place the class attributes at the top, right after the class header.

The following command prints the value of the previously declared `family` class attribute. Note that we didn't create any instance of the `TibetanSpaniel` class. Also, we specify an attribute after the class name and a dot:

```
print(TibetanSpaniel.family)
```

The following command creates an instance of the `TibetanSpaniel` class and then prints the value of the family class attribute. In this case, we will use an instance to access the class attribute:

```
brian = TibetanSpaniel("Brian", "Talking Minion", 4)
print(brian.family)
```

You can assign a new value to any class attribute. For example, the following command assigns `4` to the `obedience` class attribute:

```
TibetanSpaniel.obedience = 4
```

However, we must be very careful when we want to assign a new value to a class variable. We must address the class attribute through a class and not through an instance. If we assign a value to `obedience` through an instance named `brian`, Python will create a new instance attribute called `obedience`, instead of changing the value of the class attribute with the same name.

The following commands illustrate the problem. The first command creates a new instance attribute called `obedience` and sets its value to `8`; therefore, `brian.obedience` will be `8`. However, if we check the value of the `TibetanSpaniel.obedience` or `type(brian).obedience` class variable, the value continues to be `4`:

```
brian.obedience = 8

print(brian.obedience)
print(type(brian).obedience)
print(TibetanSpaniel.obedience)
```

Hiding data in Python using prefixes

The previously declared `TibetanSpaniel` class exposes the instance and class attributes without any kind of restriction. Thus, we can access these attributes and change their values. Python uses a special naming convention for attributes to control their accessibility. So far, we have been using attribute names without any kind of prefix; therefore, we could access attribute names within a class definition and outside of a class definition. These kinds of attributes are known as public attributes and are exposed without any restriction.

In Python, we can mark an attribute as protected by prefixing the attribute name with a leading underscore (_). For example, if we want to convert the name attribute from a public to a protected attribute, we just need to change its name from `name` to `_name`.

Whenever we mark an attribute as protected, we are telling the users of the class that they shouldn't use these attributes outside of the class definition. Thus, only the code written within the class definition and within subclasses should access attributes marked as protected. We say *should*, because Python doesn't provide a real shield for the attributes marked as protected; the language just expects users to be honest and take into account the naming convention. The following command shows a new version of the `__init__` method for the `TibetanSpaniel` class that declares three instance attributes as protected by adding a leading underscore (_) to names:

```
def __init__(self, name, favorite_toy, watchdog_ability):
    self._name = name
    self._watchdog_ability = watchdog_ability
    self._favorite_toy = favorite_toy
```

We can mark an attribute as private by prefixing the attribute name with two leading underscores (__). For example, if we want to convert the name attribute from a protected to a private attribute, we just need to change its name from `_name` to `__name`.

Whenever we mark an attribute as private, Python doesn't allow users to access the attribute outside of the class definition. The restriction also applies to subclasses; therefore, only the code written within a class can access attributes marked as private.

Python still provides access to these attributes outside of the class definition with a different name composed of a leading underscore (_), followed by the class name and the private attribute name. For example, if we use `__name` to mark name as a private attribute, it will be accessible with the `TibetanSpaniel__name` name. Obviously, the language expects users to be honest, take into account the naming convention, and avoid accessing the renamed private attribute. The following commands show a new version of the `__init__` method for the `TibetanSpaniel` class that declares three instance attributes as private by adding two leading underscores (__) to names:

```
def __init__(self, name, favorite_toy, watchdog_ability):
    self.__name = name
    self.__watchdog_ability = watchdog_ability
    self.__favorite_toy = favorite_toy
```

[The same naming convention applies to instance attributes, class attributes, instance methods, and class methods.]

Using property getters and setters in Python

Python provides a simple mechanism to define properties and specify the getter and/or setter methods. We want to make the following changes to our `TibetanSpaniel` class:

- Encapsulate the `name` attribute with a read-only `name` property
- Encapsulate the `__favorite_toy` attribute with a `favorite_toy` property
- Encapsulate the `__watchdog_ability` attribute with a `watchdog_ability` property and include the code in the setter method to assign `0` to the underlying attribute if the value specified is lower than `0` and `10` if the value specified is higher than `10`
- Define a `protection_score` read-only property with a getter method that calculates and returns a protection score based on the values of the `__watchdog_ability` private instance attribute, the `learning_rate` public class attribute, and the `problem_solving` public class attribute

We want a read-only `name` property; therefore, we just need to define a getter method that returns the value of the related `__name` private attribute. We just need to define a method named `name` and decorate it with `@property`. The following commands within the class body will do the job. Note that `@property` is included before the method's header:

```
@property
def name(self):
    return self.__name
```

After we add a getter method to define a read-only name property, we can create an instance of the edited class and try to change the value of the read-only property name, as shown in the following command:

```
merlin = TibetanSpaniel("Merlin", "Talking Smurf", 6)
merlin.name = "brian"
```

The Python console will display the following error because there is no setter method defined for the `name` property:

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: can't set attribute
```

We want to encapsulate the `__favorite_toy` private attribute with the `favorite_toy` property; therefore, we have to define both getter and setter methods. The getter method returns the value of the related `__favorite_toy` private attribute. The setter method receives the new favorite toy value as an argument and assigns this value to the related `__favorite_toy` private attribute. We have to decorate the setter method with `@favorite_toy.setter`, that is, `@`, followed by the property name and `.setter`. The following commands within the class body will do the job:

```
@property
def favorite_toy(self):
    return self.__favorite_toy

@favorite_toy.setter
def favorite_toy(self, favorite_toy):
    self.__favorite_toy = favorite_toy
```

The setter method for the favorite_toy property is very simple. The watchdog_ability property requires a setter method with more code to transform values lower than 0 in 0 and values higher than 10 in 10. The following class body will do the job:

```
@property
def watchdog_ability(self):
    return self.__watchdog_ability

@watchdog_ability.setter
def watchdog_ability(self, watchdog_ability):
    if watchdog_ability < 0:
        self.__watchdog_ability = 0
    elif watchdog_ability > 10:
        self.__watchdog_ability = 10
    else:
        self.__watchdog_ability = watchdog_ability
```

After we add the watchdog_ability property, we will create an instance of the edited class and try to set different values to this property, as shown in the following code:

```
hugo = TibetanSpaniel("Hugo", "Tennis ball", 7)
hugo.watchdog_ability = -3
print(hugo.watchdog_ability)
hugo.watchdog_ability = 30
print(hugo.watchdog_ability)
hugo.watchdog_ability = 8
print(hugo.watchdog_ability)
```

In the preceding code, after we specified -3 as the desired value for the watchdog_ability property, we printed its actual value and the result was 0. After we specified 30, the actual printed value was 10. Finally, after we specified 8, the actual printed value was 8. The code in the setter method did its job. This is how we could control the values accepted for the underlying private instance attribute.

We want a read-only protection_score property. However, this time the getter method must calculate and return a protection score based on a private instance attribute and two public class attributes. Note that the code accesses the public class attributes through type(self), followed by the attribute name. It's a safe way to access class attributes because we can change the class name or work with inheritance without unexpected issues. The following commands in the class body will do the job:

```
@property
def protection_score(self):
    return math.floor((self.__watchdog_ability + type(self).learning_
rate + type(self).problem_solving) / 3)
```

After we add the `protection_score` property, we will create an instance of the edited class and print the value of this read-only property:

```
cole = TibetanSpaniel("Cole", "Soccer ball", 4)
print(cole.protection_score)
```

Here is the complete code for the `TibetanSpaniel` class along with properties:

```
class TibetanSpaniel:
    family = "Companion, herding"
    area_of_origin = "Tibet"
    learning_rate = 9
    obedience = 3
    problem_solving = 8

    def __init__(self, name, favorite_toy, watchdog_ability):
        self.__name = name
        self.__watchdog_ability = watchdog_ability
        self.__favorite_toy = favorite_toy

    @property
    def name(self):
        return self.__name

    @property
    def favorite_toy(self):
        return self.__favorite_toy

    @favorite_toy.setter
    def favorite_toy(self, favorite_toy):
        self.__favorite_toy = favorite_toy

    @property
    def watchdog_ability(self):
        return self.__watchdog_ability

    @watchdog_ability.setter
    def watchdog_ability(self, watchdog_ability):
        if watchdog_ability < 0:
            self.__watchdog_ability = 0
        elif watchdog_ability > 10:
            self.__watchdog_ability = 10
        else:
            self.__watchdog_ability = watchdog_ability

    @property
    def protection_score(self):
        return math.floor((self.__watchdog_ability + type(self).
learning_rate + type(self).problem_solving) / 3)
```

Using methods to add behaviors to classes in Python

So far, we have added instance methods to classes and used getter and setter methods combined with decorators to define properties. Now, we want to generate a class to represent the mutable version of a 3D vector.

We will use properties with simple getter and setter methods for `x`, `y`, and `z`. The `sum` public instance method receives the delta values for `x`, `y`, and `z` and mutates an object, that is, the setter method changes the values of `x`, `y`, and `z`. Here is the initial code of the `MutableVector3D` class:

```
class MutableVector3D:
    def __init__(self, x, y, z):
        self.__x = x
        self.__y = y
        self.__z = z

    def sum(self, delta_x, delta_y, delta_z):
        self.__x += delta_x
        self.__y += delta_y
        self.__z += delta_z

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, x):
        self.__x = x

    @property
    def y(self):
        return self.__y

    @y.setter
    def y(self, y):
        self.__y = y

    @property
    def z(self):
        return self.__z

    @z.setter
    def z(self, z):
        self.__z = z
```

It's a very common requirement to generate a 3D vector with all the values initialized to `0`, that is, `x = 0`, `y = 0`, and `z = 0`. A 3D vector with these values is known as an origin vector. We can add a class method to the `MutableVector3D` class named `origin_vector` to generate a new instance of the class initialized with all the values initialized to `0`. It's necessary to add the `@classmethod` decorator before the class method header. Instead of receiving `self` as the first argument, a class method receives the current class; the parameter name is usually named `cls`. The following code defines the `origin_vector` class method:

```
    @classmethod
    def origin_vector(cls):
        return cls(0, 0, 0)
```

The preceding method returns a new instance of the current class (`cls`) with `0` as the initial value for the three elements. The class method receives `cls` as the only argument; therefore, it will be a parameterless method when we call it because Python includes a class as a parameter under the hood. The following command calls the `origin_vector` class method to generate a 3D vector, calls the sum method for the generated instance, and prints the values for the three elements:

```
mutableVector3D = MutableVector3D.origin_vector()
mutableVector3D.sum(5, 10, 15)
print(mutableVector3D.x, mutableVector3D.y, mutableVector3D.z)
```

Now, we want to generate a class to represent the immutable version of a 3D vector. In this case, we will use read-only properties for `x`, `y`, and `z`. The `sum` public instance method receives the delta values for `x`, `y`, and `z` and returns a new instance of the same class with the values of `x`, `y`, and `z` initialized with the results of the sum. Here is the code of the `ImmutableVector3D` class:

```
class ImmutableVector3D:
    def __init__(self, x, y, z):
        self.__x = x
        self.__y = y
```

```
            self.__z = z

        def sum(self, delta_x, delta_y, delta_z):
            return type(self)(self.__x + delta_x, self.__y + delta_y,
    self.__z + delta_z)

        @property
        def x(self):
            return self.__x

        @property
        def y(self):
            return self.__y

        @property
        def z(self):
            return self.__z

        @classmethod
        def equal_elements_vector(cls, initial_value):
            return cls(initial_value, initial_value, initial_value)

        @classmethod
        def origin_vector(cls):
            return cls.equal_elements_vector(0)
```

Note that the `sum` method uses `type(self)` to generate and return a new instance of the current class. In this case, the `origin_vector` class method returns the results of calling the `equal_elements_vector` class method with `0` as an argument. Remember that the `cls` argument refers to the actual class. The `equal_elements_vector` class method receives an `initial_value` argument for all the elements of the 3D vector, creates an instance of the actual class, and initializes all the elements with the received unique value. The `origin_vector` class method demonstrates how we can call another class method in a class method.

The following command calls the `origin_vector` class method to generate a 3D vector, calls the `sum` method for the generated instance, and prints the values for the three elements of the new instance returned by the `sum` method:

```
vector0 = ImmutableVector3D.origin_vector()
vector1 = vector0.sum(5, 10, 15)
print(vector1.x, vector1.y, vector1.z)
```

> As explained previously, we can change the values of the private attributes; therefore, the `ImmutableVector3D` class isn't 100 percent immutable. However, we are all adults and don't expect the users of a class with read-only properties to change the values of private attributes hidden under difficult to access names.

Encapsulating data in C#

First, we will add fields to a class in C# and then use access modifiers to hide and protect a specific member of a class from unauthorized access. We will use property getters and setters to control how we write and retrieve values to and from related fields.

We will work with auto-implemented properties to reduce the boilerplate code. We will use methods to add behaviors to classes and create the mutable and immutable version of a 3D vector to understand the difference between an object that mutates state and an object that doesn't.

Adding fields to a class

The `SmoothFoxTerrier` class is a blueprint for dogs that belong to the Smooth Fox Terrier breed. This class should inherit from a `Dog` superclass, but we will forget about inheritance and other dog breeds for a while and use the `SmoothFoxTerrier` class to understand the difference between class fields and instance fields.

We will define the following class attributes to store the values that are shared by all the members of the Smooth Fox Terrier breed. The valid values for scores are from `0` to `10`; `0` is the lowest skill and `10` the highest:

- `Family`: This is the family to which the dog breed belongs
- `AreaOfOrigin`: This is the area of origin of the dog breed
- `Energy`: This is the average energy score for the dog breed
- `ColdTolerance`: This is the average cold tolerance score for the dog breed
- `HeatTolerance`: This is the average heat tolerance score for the dog breed

The following code creates the `SmoothFoxTerrier` class and declares the previously enumerated fields and a constructor in the body of the class:

```
class SmoothFoxTerrier
{
  public static string Family = "Terrier";
  public static string AreaOfOrigin = "England";
```

```
    public static int Energy = 10;
    public static int ColdTolerance = 8;
    public static int HeatTolerance = 8;

    public string Name;
    public int WatchdogAbility;
    public string FavoriteToy;

    public SmoothFoxTerrier(string name, int watchdogAbility, string
  favoriteToy)
    {
      this.Name = name;
      this.WatchdogAbility = watchdogAbility;
      this.FavoriteToy = favoriteToy;
    }
  }
```

The preceding code initializes each class field in the same line that declares the field. The only difference between a class field and an instance field is the inclusion of the `static` keyword. This indicates that we want to create a class field.

[In C#, class fields are also known as static fields.]

The following command prints the value of the previously declared `Family` static field. Note that we didn't create any instance of the `SmoothFoxTerrier` class, and we specify a field after the class name and a dot:

```
Debug.WriteLine(SmoothFoxTerrier.Family);
```

[C# doesn't allow you to access a static field from an instance reference; therefore, we always require to use a class type to access a static field.]

You can assign a new value to any static field declared as a variable. For example, the following command assigns `8` to the `Energy` static field:

```
SmoothFoxTerrier.Energy = 8;
```

We can easily convert a static field to a read-only static field by replacing the `static` keyword with the `const` one. For example, we don't want the users of a class to change the average energy score. Therefore, we can change the line that declared the static `Energy` field with the following command that creates a static constant or read-only field:

```
public const int Energy = 10;
```

Using access modifiers

The previously declared `SmoothFoxTerrier` class exposes the instance and static fields without any kind of restriction because we declared them with the `public` access modifier. Therefore, we can access these attributes and change their values, except for the static `Energy` field that we converted to a static constant.

C# uses type member access modifiers to control which code has access to a specific type member. So far, we have been declaring all the fields with the `public` access modifier. Therefore, we could access them in a class definition and outside of a class definition.

We can use any of the following access modifiers instead of `public` to restrict access to any field:

- `protected`: C# doesn't allow users to access a field outside of the class definition. Only the code in a class or its derived classes can access the field.
- `private`: C# doesn't allow users to access a field outside of the class definition. Only the code in the class can access the field. Its derived classes cannot access the field.
- `internal`: C# doesn't allow users to access a field outside of the assembly in which a class is included. The code within the same assembly can access the field. Its derived classes cannot access the field.
- `protected internal`: C# doesn't allow users to access a field outside of the assembly in which a class is included unless it's a derived class. The code within the same assembly can access the field. Its derived classes can access the field, as happens with `protected` fields without the addition of the `internal` modifier.

The following command shows how we can change the declaration of the public `Name` field to a `protected` field. We replace the public access modifier with `protected` and the name from `Name` to `_name`. As a naming convention, we will prefix the field name with a leading underscore (_) symbol for protected or private fields:

```
protected string _name;
```

Whenever we use the `protected` access modifier in a field declaration, we restrict access to this field to the code written within the class definition and within subclasses. C# generates a real shield for the fields marked as protected; there is no way to access them outside of the explained boundaries.

The following commands show a new version of the `SmoothFoxTerrier` class that declares three instance fields as `protected` by replacing the `public` access modifier with `protected`. In addition, field names add a leading underscore (_) as a prefix and use a lowercase first letter:

```
class SmoothFoxTerrier
{
  public static string Family = "Terrier";
  public static string AreaOfOrigin = "England";
  public static int Energy = 10;
  public static int ColdTolerance = 8;
  public static int HeatTolerance = 8;

  protected string _name;
  protected int _watchdogAbility;
  protected string _favoriteToy;

  public SmoothFoxTerrier(string name, int watchdogAbility, string
favoriteToy)
  {
    this._name = name;
    this._watchdogAbility = watchdogAbility;
    this._favoriteToy = favoriteToy;
  }
}
```

The following code shows how we can change the declaration of the `_name` protected field to a `private` field. We replace the `protected` access modifier with `private`:

```
private string _name;
```

Whenever we use the `private` access modifier in a field declaration, we restrict access to the field to the code written in the class definition and subclasses. C# generates a real shield for the fields marked as `private`; there is no way to access these outside of the class definition. The restriction also applies to subclasses; therefore, only the code written in a class can access attributes marked as private.

The following commands show a new version of the instance fields declaration for the `SmoothFoxTerrier` class that declares the three instance fields as `private`, replacing the `protected` keyword with `private`:

```
private string _name;
private int _watchdogAbility;
private string _favoriteToy;
```

After the previous changes are done, it's necessary to replace the code in the constructor with the following commands. This helps initialize new field names:

```
public SmoothFoxTerrier(string name, int watchdogAbility, string
favoriteToy)
{
  this._name = name;
  this._watchdogAbility = watchdogAbility;
  this._favoriteToy = favoriteToy;
}
```

[We can use the same access modifiers for any type of member, including instance fields, static fields, instance methods, and static methods.]

Using property getters and setters in C#

C# provides a simple yet powerful mechanism to define properties and specify the getter method and/or the setter method. We want to make the following changes to our `SmoothFoxTerrier` class:

- Encapsulate the `_name` field with a read-only `Name` property.
- Encapsulate the `_favoriteToy` attribute with a `FavoriteToy` property.
- Encapsulate the `_watchdogAbility` attribute with a `WatchdogAbility` property and include the code in the setter method to assign `0` to the underlying attribute if the value specified is lower than `0` and assign `10` when the value specified is higher than `10`.
- Define a `ProtectionScore` read-only property with a getter method that calculates and returns a protection score based on the values of the `_watchdogAbility` private instance field, the `ColdTolerance` public static field, and the `HeatTolerance` public static field.

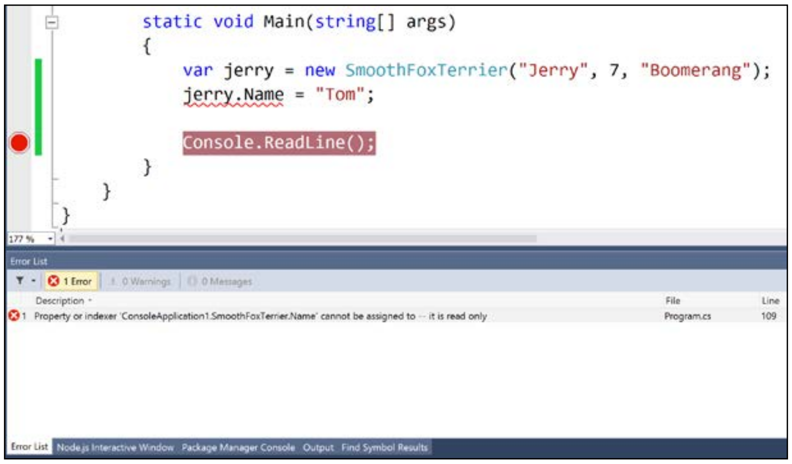
We want a read-only `Name` property; therefore, we just need to define a getter method that returns the value of the related `_name` private field. We just need to add some code in the class body. We declare the `Name` property as a `public` field of the `string` type, followed by the definition of the getter method enclosed within a pair of curly braces ({ }). This encloses the definition of the getter method and/or the setter method. In this case, we just need a getter method. The `get` keyword starts the definition of the getter method whose contents are enclosed within a pair of curly braces ({ }). The following code in the class body will do the job:

```
public string Name
{
  get
  {
    return _name;
  }
}
```

After we add a getter method to define a read-only `Name` property, we will create an instance of the edited class and write code that changes the value of the `Name` read-only property, as shown in the following command:

```
var jerry = new SmoothFoxTerrier("Jerry", 7, "Boomerang");
jerry.Name = "Tom";
```

The code that assigns a value to the read-only `Name` property won't allow the console application to compile. It will display a build error because the compiler cannot find a setter method declared for the `Name` property. The specific error message is `Property or indexer 'ConsoleApplication1.SmoothFoxTerrier.Name' cannot be assigned to -- it is read only`. The following screenshot shows the generated error in the IDE:



```
        static void Main(string[] args)
        {
            var jerry = new SmoothFoxTerrier("Jerry", 7, "Boomerang");
            jerry.Name = "Tom";

            Console.ReadLine();
        }
    }
}
```

Error List

	Description	File	Line
1	Property or indexer 'ConsoleApplication1.SmoothFoxTerrier.Name' cannot be assigned to -- it is read only	Program.cs	109

We want to encapsulate the `_favoriteToy` private field with a `FavoriteToy` property; therefore, we have to define the getter method and the setter method. The getter method returns the value of the related `_favoriteToy` private field. The setter method receives the new favorite toy value as an argument named `value` and assigns this value to the related __`favoriteToy` private field. We declare the `FavoriteToy` property as a `public` field of the `string` type, followed by the definition of the getter method and the setter method enclosed in a pair of curly braces ({ }). The `get` keyword starts the definition of the getter method whose contents are enclosed within a pair of curly braces ({ }). The `set` keyword starts the definition of the setter method whose contents are enclosed within a pair of curly braces ({ }). The `set` keyword doesn't declare an argument named `value`. However, the code in this method receives the value assigned to the property in an implicit argument named `value`. The following code in the class body will do the job:

```
public string FavoriteToy
{
  get
  {
    return this._favoriteToy;
  }

  set
  {
    this._favoriteToy = value;
  }
}
```

The setter method for the `FavoriteToy` property is very simple because it just assigns the specified value to the related private field. The `WatchdogAbility` property requires a setter method with more code to transform values lower than `0` to `0` and values higher than `10` to `10`. The following code in the class body will do the job:

```
public string FavoriteToy
{
  get
  {
    return this._favoriteToy;
  }

  setpublic int WatchdogAbility
  {
    get
    {
      return this._watchdogAbility;
```

```
        }
        set
        {
          if (value < 0) {
            this._watchdogAbility = 0;
          } else if (value > 10) {
            this._watchdogAbility = 10;
          } else {
            this._watchdogAbility = value;
          }
        }
      }

      {
        this._favoriteToy = value;
      }
    }
```

After we add the `WatchdogAbility` property, we will create an instance of the edited class and try to set different values to this property, as shown in the following code:

```
var tom = new SmoothFoxTerrier("Tom", 8, "Boomerang");
tom.WatchdogAbility = -9;
Console.WriteLine(tom.WatchdogAbility);
tom.WatchdogAbility = 52;
Console.WriteLine(tom.WatchdogAbility);
tom.WatchdogAbility = 9;
Console.WriteLine(tom.WatchdogAbility);
```

After we specified `-9` as the desired value for the `WatchdogAbility` property, we printed its actual value to the console and the result was `0`. After we specified `52`, the actual printed value was `10`. Finally, after we specified `9`, the actual printed value was `9`. The code in the setter method did its job; we can control all the values accepted for the underlying private instance field.

We want a read-only `ProtectionScore` property. However, this time, the getter method must calculate and return a protection score based on a private instance field and two public static fields. Note that the code accesses all the public class fields through `SmoothFoxTerrier`, followed by the static field name. The following code in the class body will do the job:

```
public int ProtectionScore
{
  get
  {
      return (int)Math.Floor ((this._watchdogAbility + SmoothFoxTerrier.
ColdTolerance + SmoothFoxTerrier.HeatTolerance) / 3d);
    }
  }
```

After we add the `ProtectionScore` property, we will create an instance of the edited class and print the value of this read-only property to the console:

```
var laura = new SmoothFoxTerrier("Laura", "Old sneakers", 9);
console.WriteLine(laura.ProtectionScore);
```

The following code shows the complete code for the `SmoothFoxTerrier` class with its properties:

```
class SmoothFoxTerrier
{
  public static string Family = "Terrier";
  public static string AreaOfOrigin = "England";
  public const int Energy = 10;
  public static int ColdTolerance = 8;
  public static int HeatTolerance = 8;

  private string _name;
  private int _watchdogAbility;
  private string _favoriteToy;

  public string Name
  {
    get
    {
      return this._name;
    }
  }

  public string FavoriteToy
  {
    get
    {
      return this._favoriteToy;
    }

    set
    {
      this._favoriteToy = value;
    }
  }

  public int WatchdogAbility
  {
    get
    {
      return this._watchdogAbility;
    }
    set
    {
      if (value < 0) {
        this._watchdogAbility = 0;
      } else if (value > 10) {
        this._watchdogAbility = 10;
      } else {
        this._watchdogAbility = value;
      }
    }
  }

  public int ProtectionScore
  {
    get
    {
      return Math.Floor ((this._watchdogAbility + SmoothFoxTerrier.
ColdTolerance + SmoothFoxTerrier.HeatTolerance) / 3);
    }
  }

  public SmoothFoxTerrier(string name, int watchdogAbility, string
favoriteToy)
  {
    this._name = name;
    this._watchdogAbility = watchdogAbility;
    this._favoriteToy = favoriteToy;
  }
}
```

Working with auto-implemented properties

When we don't require any specific logic in the getter method or the setter method, we can take advantage of a simplified mechanism to declare properties called auto-implemented properties. For example, the following code use auto-implemented properties to simplify the declaration of the `FavoriteToy` property:

```
public string FavoriteToy { get; set; }
```

The previous code declares the `FavoriteToy` property with empty getter and setter methods. The compiler creates a private and anonymous field related to the defined property that only the property's automatically generated getters and setters access. If you need to customize either the getter method or the setter method in the future, you can replace the usage of auto-implemented properties with specific getter and setter methods and add your own private field to support a property if necessary.

If we use the previously defined `FavoriteToy` property with auto-implemented properties, we have to remove the `_favoriteToy` private field. In addition, we have to assign the `favoriteToy` private field received as an argument in the constructor to the `FavoriteToy` property instead of working with the private field, as shown in the following code:

```
this.FavoriteToy = favoriteToy;
```

We can also use auto-implemented properties when we want to create a read-only property that doesn't require any specific logic in the getter method. For example, the following code uses auto-implemented properties to simplify the declaration of the `Name` property:

```
public string Name { get; private set; }
```

The preceding code declares the setter method, also known as the **set accessor**, as private. This way, `Name` is a read-only property. As in the previous example, if we use the previously defined `Name` property with auto-implemented properties, we have to remove the `_name` private field. In addition, we have to assign the `name` property received as an argument in the constructor to the `Name` property instead of working with the private field, as shown in the following line of code:

```
this.Name = name;
```

The setter method is private; therefore, we can set the value for `Name` in the class. However, we cannot access the setter method outside of this class; therefore, it becomes a read-only property.

Using methods to add behaviors to classes in C#

So far, we have added instance methods to a class in C#, and used getter and setter methods to define properties. Now, we want to generate a class to represent the mutable version of a 3D vector in C#.

We will use auto-implemented properties for `X`, `Y`, and `Z`. The public `Sum` instance method receives the delta values for `X`, `Y`, and `Z` (`deltaX`, `deltaY`, and `deltaZ`) and mutates the object, that is, the method changes the values of `X`, `Y`, and `Z`. The following shows the initial code of the `MutableVector3D` class:

```
class MutableVector3D
{
  public double X { get; set; }
  public double Y { get; set; }
  public double Z { get; set; }

  public void Sum(double deltaX, double deltaY, double deltaZ)
  {
    this.X += deltaX;
    this.Y += deltaY;
    this.Z += deltaZ;
  }

  public MutableVector3D(double x, double y, double z)
  {
    this.X = x;
    this.Y = y;
    this.Z = z;
  }
}
```

It's a very common requirement to generate a 3D vector with all the values initialized to `0`, that is, `X = 0`, `Y = 0`, and `Z = 0`. A 3D vector with these values is known as an origin vector. We can add a class method to the `MutableVector3D` class named `OriginVector` to generate a new instance of the class initialized with all the values initialized to `0`. Class methods are also known as static methods in C#. It's necessary to add the `static` keyword after the public access modifier before the class method name. The following commands define the `OriginVector` static method:

```
public static MutableVector3D OriginVector()
{
  return new MutableVector3D(0, 0, 0);
}
```

The preceding method returns a new instance of the `MutableVector3D` class with `0` as the initial value for all the three elements. The following code calls the `OriginVector` static method to generate a 3D vector, calls the `Sum` method for the generated instance, and prints the values for all the three elements on the console output:

```
var mutableVector3D = MutableVector3D.OriginVector();
mutableVector3D.Sum(5, 10, 15);
Console.WriteLine(mutableVector3D.X, mutableVector3D.Y,
mutableVector3D.Z)
```

Now, we want to generate a class to represent the immutable version of a 3D vector. In this case, we will use read-only properties for `X`, `Y`, and `Z`. We will use auto-generated properties with `private set`. The `Sum` public instance method receives the delta values for `X`, `Y`, and `Z` (`deltaX`, `deltaY`, and `deltaZ`) and returns a new instance of the same class with the values of `X`, `Y`, and `Z` initialized with the results of the sum. The code for the `ImmutableVector3D` class is as follows:

```
class ImmutableVector3D
{
  public double X { get; private set; }
  public double Y { get; private set; }
  public double Z { get; private set; }

  public ImmutableVector3D Sum(double deltaX, double deltaY, double
deltaZ)
  {
    return new ImmutableVector3D (
    this.X + deltaX,
    this.Y + deltaY,
    this.Z + deltaZ);
  }

  public ImmutableVector3D(double x, double y, double z)
  {
    this.X = x;
    this.Y = y;
    this.Z = z;
  }

  public static ImmutableVector3D EqualElementsVector(double
initialValue)
  {
    return new ImmutableVector3D(initialValue, initialValue,
initialValue);
```

```
    }

    public static ImmutableVector3D OriginVector()
    {
      return ImmutableVector3D.EqualElementsVector(0);
    }
  }
```

In the new class, the `Sum` method returns a new instance of the `ImmutableVector3D` class, that is, the current class. In this case, the `OriginVector` static method returns the results of calling the `EqualElementsVector` static method with `0` as an argument. The `EqualElementsVector` class method receives an `initialValue` argument for all the elements of the 3D vector, creates an instance of the actual class, and initializes all the elements with the received unique value. The `OriginVector` static method demonstrates how we can call another static method in a static method.

The following code calls the `OriginVector` static method to generate a 3D vector, calls the `Sum` method for the generated instance, and prints all the values for the three elements of the new instance returned by the `Sum` method on the console output:

```
var vector0 = ImmutableVector3D.OriginVector();
var vector1 = vector0.Sum(5, 10, 15);
Console.WriteLine(vector1.X, vector1.Y, vector1.Z);
```

C# doesn't allow users of the `ImmutableVector3D` class to change the values of `X`, `Y`, and `Z` properties. The code doesn't compile if you try to assign a new value to any of these properties. Thus, we can say that the `ImmutableVector3D` class is 100 percent immutable.

Encapsulating data in JavaScript

First, we will add properties to a constructor function in JavaScript. Then, we will use local variables to hide and protect specific members of a class from unauthorized access. We will use property getters and setters to control how we write and retrieve values to and from related local variables.

We will use methods to add behaviors to objects. Also, we will create the mutable and immutable version of a 3D vector to understand the difference between an object that mutates state and an object that doesn't.

Adding properties to a constructor function

As it so happens with dogs, cats also have breeds. The `ScottishFold` constructor function provides a blueprint for cats that belong to the Scottish Fold breed. We will use the `ScottishFold` constructor function to understand how we can take advantage of the fact that a constructor function is an object in JavaScript.

As it so happens with any other cat breeds, Scottish Fold cats have some profile values. We will add the following properties to the constructor function to store the values that are shared by all the members of the Scottish Fold breed. Note that the valid values for scores range from `0` to `5`; `0` is the lowest skill and `5` the highest:

- `generalHealth`: This is a score based on the genetic illnesses that are common to the cat breed
- `affectionateWithFamily`: This is a score based on the probability for the cat to shower the whole family with affection
- `intelligence`: This is a score based on how smart the cats that belong to this breed are
- `kidFriendly`: This is a score based on how tolerant of children the cats that belong to this breed are
- `petFriendly`: This is a score based on the likeliness of the cats that belong to this breed are to accept other pets at home

The following code defines a `ScottishFold` constructor function and then adds the previously enumerated attributes as properties of the constructor function:

```
function ScottishFold(name, favoriteToy, energy) {
  this.name = name;
  this.favoriteToy = favoriteToy;
  this.energy = energy;
}

ScottishFold.generalHealth = 3;
ScottishFold.affectionateWithFamily = 5;
ScottishFold.intelligence = 4;
ScottishFold.kidFriendly = 5;
ScottishFold.petFriendly = 4;
```

The preceding code assigns a value to properties of the `ScottishFold` constructor function after the definition of the constructor function. The following line of code prints the value of the previously declared `generalHealth` property. Note that we didn't use the `ScottishFold` constructor function to create any instance; we specify an attribute after the constructor function name and a dot:

```
console.log(ScottishFold.generalHealth)
```

You can assign a new value to any constructor function property. For example, the following line of code assigns the value `4` to the `generalHealth` property:

```
ScottishFold.generalHealth = 4;
```

The following lines of code creates an instance with the `ScottishFold` constructor function and then prints the value of the `generalHealth` property. In this case, we can use an instance to access a property through the constructor function stored in the `constructor` property:

```
Var lucifer = new ScottishFold("Lucifer", "Tennis ball", 4);
console.log(lucifer.constructor.generalHealth);
```

Hiding data in JavaScript with local variables

The previously declared `ScottishFold` constructor function generates instances that expose the instance and constructor function properties without any kind of restriction. Thus, we can access these properties and change their values.

JavaScript doesn't provide access modifiers. However, we can declare local variables in constructor functions to protect them from being accessed outside of the object blueprint definition. Only functions that are declared in the constructor function will be able to access local variables. The following code shows a new version of the `ScottishFold` constructor function that declares three local variables using the `var` keyword instead of the `this.` prefix. In addition, the following code adds a leading underscore (_) to the names:

```
function ScottishFold(name, favoriteToy, energy) {
  var _name = name;
  var _favoriteToy = favoriteToy;
  var _energy = energy;
}
```

The constructor function saves the values of all the three arguments in local variables. Thus, if we create a `ScottishFold` instance, we won't be able to access these variables. The following code tries to retrieve the value of all the local variables from a `ScottishFold` instance. The three lines that use all the variable names display undefined because there is no property with specified names. Only functions that are defined in the constructor function can access all the local variables:

```
Var lucifer = new ScottishFold("Lucifer", "Tennis ball", 4);
console.log(lucifer._name);
console.log(lucifer._favoriteToy);
console.log(lucifer._energy);
```

Using property getters and setters in JavaScript

JavaScript provides a mechanism to specify the getter method and/or setter method for properties. We want to make the following changes to our `ScottishFold` class:

- Encapsulate the `_name` local variable with a read-only `name` property
- Encapsulate the `_favoriteToy` local variable with a `favoriteToy` property
- Encapsulate the `_energy` local variable with an `energy` property and include a code in the setter method to assign `0` to the underlying attribute if the value specified is lower than `0`, and assign `5` if the value specified is higher than `5`

We want a read-only `name` property; therefore, we just need to define a getter method that returns the value of the related `_name` local variable. We just need to add some code in the class function declaration:

```
function ScottishFold(name, favoriteToy, energy) {
  var _name = name;
  var _favoriteToy = favoriteToy;
  var _energy = energy;

Object.defineProperty(this, 'name', { get: function(){ return _name; }
});
}
```

The code calls the `Object.defineProperty` function with the following arguments:

- `this`: This is the instance.
- `'name'`: This is the desired name for the property as a string.
- This is an object with the getter function code specified in the `get` property. Note that the getter function can access the `_name` variable defined in the constructor function.

After we add a getter method to define a read-only `name` property, we can create an instance of the edited constructor function and write code that reads, tries to change, and reads again the value of the `name` read-only property, as shown in the following code:

```
var lucifer = new ScottishFold("Lucifer", "Tennis ball", 4);
console.log(lucifer.name);
lucifer.name = "Jerry";
console.log(lucifer.name);
```

After we change the value of the `name` property, we can print the value of this property on the JavaScript console. It's still the same value that we specified when calling the `Lucifer` constructor function. There is no setter method defined; therefore, `name` is a read-only property.

We want to encapsulate the `_favoriteToy` local variable with a `favoriteToy` property; therefore, we have to define getter and setter methods. The getter method returns the value of the related `_favoriteToy` local variable. The setter method receives the new favorite toy value as a `val` argument named and assigns this value to the related `_favoriteToy` local variable. The following code in the class function declaration will do the job:

```
function ScottishFold(name, favoriteToy, energy) {
  var _name = name;
  var _favoriteToy = favoriteToy;
  var _energy = energy;

  Object.defineProperty(this, 'name', { get: function(){ return _name;
} });
    Object.defineProperty(this, 'favoriteToy', { get: function(){
return _favoriteToy; }, set: function(val){ _favoriteToy = val; } });
}
```

The setter method for the `favoriteToy` property is very simple because it just assigns the specified value to the related local variable. The `energy` property requires a setter method with more code to transform values lower than `0` to `0` and values higher than `5` to `5`. The following code in the constructor function will do the job:

```
function ScottishFold(name, favoriteToy, energy) {
  var _name = name;
  var _favoriteToy = favoriteToy;
  var _energy = energy;

  Object.defineProperty(this, 'name', { get: function(){ return _name;
} });
```

```
    Object.defineProperty(this, 'favoriteToy', { get: function(){ return
  _favoriteToy; }, set: function(val){ _favoriteToy = val; } });
    Object.defineProperty(
    this,
    'energy', {
      get: function(){ return _energy; },
      set: function(val){
        if (val < 0) {
          _energy = 0;
        } else if (val > 5) {
          _energy = 5;
        } else {
          _energy = val;
        }
      }
    });
  }
```

After we add the `energy` property, we can create an instance of the edited class and try to set different values to this property, as shown in the following code:

```
var garfield = new ScottishFold("Garfield", "Pillow", 1);
garfield.energy = -7;
console.log(Garfield.energy);
garfield.energy = 35;
console.log(Garfield.energy);
garfield.energy = 3;
console.log(Garfield.energy);
```

In the preceding code, after we specified `-7` as the desired value for the `energy` property, we printed its actual value to the console. The result was `0`. After we specified `35`, the actual printed value was `5`. Finally, after we specified `3`, the actual printed value was `3`. The code in the setter method did its job; we could control the values accepted for the underlying local variable.

Using methods to add behaviors to constructor functions

So far, we have added methods to a constructor function that produced instance methods in a generated object. In addition, we used getter and setter methods combined with local variables to define properties. Now, we want to generate a constructor function to represent the mutable version of a 3D vector.

We will use properties with simple getter and setter methods for `x`, `y`, and `z`. The `sum` public instance method receives the delta values for `x`, `y`, and `z` and mutates an object, that is, the method changes the values of `x`, `y`, and `z`. The following code shows the initial code of the `MutableVector3D` constructor function:

```
function MutableVector3D(x, y, z) {
  var _x = x;
  var _y = y;
  var _z = z;

  Object.defineProperty(this, 'x', {
    get: function(){ return _x; },
    set: function(val){ _x = val; }
  });

  Object.defineProperty(this, 'y', {
    get: function(){ return _y; },
    set: function(val){ _y = val; }
  });

  Object.defineProperty(this, 'z', {
    get: function(){ return _z; },
    set: function(val){ _z = val; }
  });

  this.sum = function(deltaX, deltaY, deltaZ) {
    _x += deltaX;
    _y += deltaY;
    _z += deltaZ;
  }
}
```

It's a very common requirement to generate a 3D vector with all the values initialized to `0`, that is, `x = 0`, `y = 0`, and, `z = 0`. A 3D vector with these values is known as an origin vector. We can add a function to the `MutableVector3D` constructor function named `originVector` to generate a new instance of a class with all the values initialized to `0`. The following code defines the `originVector` function:

```
MutableVector3D.originVector = function() {
  return new MutableVector3D(0, 0, 0);
};
```

The method returns a new instance built in the `MutableVector3D` constructor function with `0` as the initial value for all the three elements. The following code calls the `originVector` function to generate a 3D vector, calls the `sum` method for the generated instance, and prints all the values for all the three elements:

```
var mutableVector3D = MutableVector3D.originVector();
mutableVector3D.sum(5, 10, 15);
console.log(mutableVector3D.x, mutableVector3D.y, mutableVector3D.z);
```

Now, we want to generate a constructor function to represent the immutable version of a 3D vector. In this case, we will use read-only properties for `x`, `y`, and `z`. In this case, we will use the `ImmutableVector3D.prototype` property to define the `sum` method. The method receives the values of delta for `x`, `y`, and `z`, and returns a new instance with the values of `x`, `y`, and `z` initialized with the results of the sum. The following code shows the `ImmutableVector3D` constructor function and the additional code that defines all the other methods:

```
function ImmutableVector3D(x, y, z) {
  var _x = x;
  var _y = y;
  var _z = z;

  Object.defineProperty(this, 'x', {
    get: function(){ return _x; }
  });

  Object.defineProperty(this, 'y', {
    get: function(){ return _y; }
  });

  Object.defineProperty(this, 'z', {
    get: function(){ return _z; }
  });
}

ImmutableVector3D.prototype.sum = function(deltaX, deltaY, deltaZ) {
  return new ImmutableVector3D(
  this.x + deltaX,
  this.y + deltaY,
  this.z + deltaZ);
};

ImmutableVector3D.equalElementsVector = function(initialValue) {
  return new ImmutableVector3D(initialValue, initialValue,
initialValue);
};

ImmutableVector3D.originVector = function() {
  return ImmutableVector3D.equalElementsVector(0);
};
```

Again, note that the preceding code defines the `sum` method in the `ImmutableVector3D.prototype` method. This method will be available to all the instances generated in the `ImmutableVector3D` constructor function. The `sum` method generates and returns a new instance of `ImmutableVector3D`. In this case, the `originVector` method returns the results of calling the `equalElementsVector` method with `0` as an argument. The `equalElementsVector` method receives an `initialValue` argument for all the elements of the 3D vector, creates an instance of the actual class, and initializes all the elements with the received unique value. The `originVector` method demonstrates how we can call another function defined in the constructor function.

The following code calls the `originVector` method to generate a 3D vector, calls the `sum` method for the generated instance, and prints the values for all the three elements of the new instance returned by the `sum` method:

```
var vector0 = ImmutableVector3D.originVector();
var vector1 = vector0.sum(5, 10, 15);
console.log(vector1.x, vector1.y, vector1.z);
```

In this case, we took advantage of the `prototype` property. We will dive deeper into the advantages of how to work with the prototype property through out the course of this book.

Summary

In this chapter, we looked at the different members of a class or a blueprint. We worked with naming conventions in Python to hide attributes, took advantage of access modifiers in C#, and worked with local variables in a constructor function in JavaScript. We declared properties in different programming languages and customized their getter and setter methods.

We worked with dogs and cats and defined the shared properties of their breeds in classes and constructor functions. We also worked with mutable and immutable versions of a 3D vector.

Now that you have learned how to encapsulate data, we are ready to work with inheritance and specialization in Python, JavaScript, and C#, which are the topics of the next chapter.

4
Inheritance and Specialization

In this chapter, we will create a hierarchy of blueprints that generate objects. We will take advantage of inheritance and many related features to specialize behavior in each of the three covered programming languages. We will:

- Use classes to abstract behavior
- Understand the concept of simple inheritance and design a hierarchy of classes
- Learn the difference between overloading and overriding methods
- Understand the concept of overloading operators
- Understand polymorphism
- Take advantage of the prototype chain to use inheritance in JavaScript

Using classes to abstract behavior

So far, we have been creating classes on Python and C# to generate blueprints for real-life objects. In JavaScript we have been using constructor functions to achieve the same goal. Now it is time to take advantage of more advanced features of object-oriented programming and start designing a hierarchy of classes instead of working with isolated classes. Based on our requirements we will first design all the classes that we need. Then we will use all the features available in each of the covered programming languages to code the design.

We worked with dogs, cats, and some of their breeds. Now let's imagine that we have to work with a more complex solution that requires us to work with hundreds of breeds. In addition, we already know that our application will start working with domestic cats and dogs, but in the future it will be necessary to work with other members of the cat family, other mammals, other domestic mammals, reptiles, and birds. Thus, our object-oriented design needs to be ready for expansion purposes if required. However, wait! The animal kingdom is extremely complex and we don't want to model a complete representation of the animal kingdom and its classification; we just want to create all the necessary classes to have a flexible model that can be easily expanded.

So, this time a few classes won't be enough to represent the breeds of cats and dogs. The following list enumerates the classes that we will create along with their descriptions:

- `Animal`: This is an abstract class that generalizes all the members of the animal kingdom. Dogs, cats, reptiles, and birds have one thing in common: they are animals. Thus it makes sense to create an abstract class that will be the baseline for all the different classes of animals that we will have to represent in our object-oriented design.
- `Mammal`: This is a class that generalizes mammals. They are different from reptiles, amphibians, birds, and insects. We already know that we will also have to model reptiles and birds; therefore, we will create a `Mammal` class at this level.
- `DomesticMammal`: The tiger (*Panthera tigris*) is the largest and heaviest living species of the cat family. A tiger is a cat but it is completely different from a domestic cat. Our initial requirements tell us that we will work with domestic and wild animals; therefore, we will create a class that generalizes domestic mammals. In the future, we will have `WildMammal` that will generalize wild mammals.
- `Dog`: We could go on specializing the `DomesticMammal` class with additional subclasses until we reach a `Dog` class. For example, we can create a `CanidCarnivorianDomesticMammal` subclass and then make the `Dog` class inherit from it. However, the kind of application we have to develop doesn't require any intermediary classes between `DomesticMammal` and `Dog`. At this level, we will also have a `Cat` class. The `Dog` class generalizes the properties and methods required for a dog in our application. Subclasses of the `Dog` class will represent the different families of the dog breeds. For example, one of the main differences between a dog and a cat in our application domain is that a dog barks and a cat meows.

- `TerrierDog`: Each dog breed belongs to a family. We will work with a large number of dog breeds, and some profile values determined by their family are very important for our application. Thus we will create a subclass of the `Dog` class for each family. In this case, the sample `TerrierDog` class represents the Terrier family.
- `SmoothFoxTerrier`: Finally, a subclass of the dog breed family class will represent a specific dog breed that belongs to a family. Its breed determines the dog's look and behavior. A dog that belongs to the Smooth Fox Terrier breed will look and behave completely different than a dog that belongs to the Tibetan Spaniel breed. Thus we will create instances of all the classes at this level to give life to each dog in our application. In this case, the `SmoothFoxTerrier` class models an animal, a mammal, a domestic mammal, a dog, and a terrier family dog, specifically, a dog that belongs to the Smooth Fox Terrier breed.

Each class listed in the preceding list represents a specialization of the previous class, that is, its superclass, parent class, or superset, as shown in the following table:

Superclass, parent class, or superset	**Subclass, child class, or subset**
`Animal`	`Mammal`
`Mammal`	`DomesticMammal`
`DomesticMammal`	`Dog`
`Dog`	`TerrierDog`
`TerrierDog`	`SmoothFoxTerrier`

Our application requires many members of the Terrier family; therefore, the `SmoothFoxTerrier` class isn't going to be the only subclass of `TerrierDog`. In the future, we will have the following three additional subclasses of `TerrierDog`:

- `AiredaleTerrier`: This subclass represents the Airedale Terrier breed
- `BullTerrier`: This subclass represents the Bull Terrier breed
- `CairnTerrier`: This subclass represents the Cairn Terrier breed

Understanding inheritance

When a class inherits from another class, it inherits all the elements that compose the parent class, also known as superclass. The class that inherits all the elements of the parent class is known as a subclass. For example, the `Mammal` subclass inherits all the properties, instance fields or attributes, and class fields or attributes defined in the `Animal` superclass.

> You don't have to forget what you learned in *Chapter 3, Encapsulation of Data*, about access modifiers and naming conventions that restrict access to certain members. We must take them into account to determine the inherited members that we will be able to access in subclasses. Some access modifiers and naming conventions applied to members don't allow subclasses to access these members defined in superclasses.

The `Animal` abstract class is the baseline for our class hierarchy. We require each animal to specify its age; therefore, we will have to specify the age of the animal when we create an instance of the `Animal` class. This class will define an `age` property and display a message whenever an instance of an animal has been created. The `Animal` class defines two attributes that specify the number of legs and pair of eyes. Both these attributes will be initialized to `0`, but its subclasses will have to set a value for these attributes. The `Animal` class defines two instance methods:

- **Print legs and eyes**: This method prints the number of legs and eyes of an animal
- **Print age**: This method prints an animal's age

In addition, we want to be able to compare the age of the different `Animal` instances using the following comparison operators when the programming language allows you to do it. The following are the comparison operators:

- Less than (`<`)
- Less than or equal to (`<=`)
- Greater than (`>`)
- Greater than or equal to (`>=`)

If your programming language doesn't allow you to use the previously enumerated operators to compare the age of the `Animal` instances, we can define instance methods with their appropriate names in the `Animal` class to achieve the same goal.

Wait! We said that we had to print a message whenever we created an `Animal` instance. However, we want animal to be an abstract class; therefore, we aren't supposed to create instances of this class. Thus, it seems that it is completely impossible to achieve our goal. When we inherit from a class, we also inherit its constructor; therefore, we can call the inherited constructor to run the initialization code for the base class. This way, it is possible to know when an instance of `Animal` is being created even when it is an abstract class. In fact, all the instances of subclasses of the `Animal` class are going to be instances of `Animal` too.

The `Mammal` abstract class inherits from the `Animal` superclass and specifies `1` as the value for a pair of eyes. The `Animal` superclass defines this class attribute with `0` as the initial value, but the `Mammal` subclass overwrites it with `1`. So far, all the mammals discovered on earth have just one pair of eyes. If scientists discover evidence of a mammal with more than one pair of eyes, we don't need this weird animal in our application; therefore, we won't worry about it.

We require each mammal to specify its age and whether it is pregnant when you create an instance of a mammal. The `Mammal` class inherits the age property from the `Animal` superclass; therefore, it is only necessary to add a property that allows you to access the `is pregnant` attribute. Note that we don't specify the gender at any time in order to keep things simple. If we add a gender attribute, we will need a validation to avoid a male gender being pregnant. Right now, our focus is on inheritance. The `Mammal` class displays a message whenever a mammal is created.

> Each class inherits from one class; therefore, each new class that we will define has just one superclass. In this case, we will always work with *single inheritance*.

The `DomesticMammal` abstract class inherits from the `Mammal` class. We require each `DomesticMammal` abstract class to specify its name and favorite toy. Any domestic mammal has a name; it always picks a favorite toy. Sometimes, the favorite toy is not exactly the toy we would like them to pick (our shoes or sneakers), but let's keep the focus on our classes. It is necessary to add a read-only property that allows you to access the name attribute and the read/write property for the favorite toy. You cannot change the name of the domestic mammal, but you can force the mammal to change its favorite toy. The `DomesticMammal` class displays a message whenever a domestic mammal is created.

The `talk` instance method will display a message. This message indicates that the domestic mammal name is concatenated with the word `talk`. Each subclass must make the specific domestic mammal talk in a different way. A parrot can really talk but we will consider a dog's bark and a cat's meow as if they were talking.

The `Dog` class inherits from `DomesticMammal` and specifies `4` as the value for the number of legs. The `Animal` class, that is the `Mammal` superclass, defined this class attribute with `0` as its value, but `Dog` overwrites this inherited attribute with `4`. The `Dog` class displays a message whenever a dog instance is created.

We want dogs to be able to bark; therefore, we need a `bark` method. The `bark` method has to allow a dog to perform the following things:

- Bark happily just once
- Bark happily a specific number of times
- Bark happily to another domestic mammal with a name just once
- Bark happily to another domestic mammal with a name a specific number of times
- Bark angrily just once
- Bark angrily a specific number of times
- Bark angrily to another domestic mammal with a name just once
- Bark angrily to another domestic mammal with a name a specific number of times

We can have just one `bark` method or many `bark` methods. There are different mechanisms to solve the challenges of the different ways in which a dog must be able to bark. Not all the programming languages support the same mechanisms introduced in the classic object-oriented programming approach.

When we call the `talk` method for any dog, we want it to bark happily once. We don't want to display the message defined in the `talk` method introduced in the `DomesticMammal` class.

We want to know the breed and the breed family to which a dog belongs. Thus, we will define the dog's breed and breed family class attributes. Each subclass of the `Dog` superclass must specify the appropriate value for these class attributes. In addition, the two class methods will allow you to print the dog's breed and the dog's breed family.

The `TerrierDog` class inherits from `Dog` and specifies `Terrier` as the value for the breed family. This class displays a message whenever a `TerrierDog` class has been created.

Finally, the `SmoothFoxTerrier` class inherits from `TerrierDog` and specifies `Smooth Fox Terrier` as the value for the dog's breed. The `SmoothFoxTerrier` class displays a message whenever a `SmoothFoxTerrier` class has been created.

Understanding method overloading and overriding

Some programming languages allow you to define a method with the same name multiple times by passing different arguments. This feature is known as **method overloading**. In some cases, we can overload a constructor. However, it is very important to mention that a similar effect can be achieved with optional parameters or default values for specific arguments.

For example, we can take advantage of method overloading in a programming language that supports it to define multiple instances of the `bark` method. However, it is very important to avoid code duplication when we overload methods.

Sometimes, we define a method in a class and know that a subclass may need to provide a different instance of this method. When a subclass provides a different implementation of a method defined in a superclass with the same name, same arguments, and same return type, we say that we have overridden a method. When we override a method, the implementation in the subclass overwrites the code given in the superclass.

It is also possible to override properties and other members of a class in subclasses.

Understanding operator overloading

Some programming languages, such as C# and Python, allow you to redefine specific operators to work in a different way based on the classes in which we apply them. For example, we can use comparison operators—such as less than (`<`) and greater than (`>`)—to return the results of comparing the age value when they are applied to instances of `Dog`.

The redefinition of operators to work in a specific way when applied to instances of specific classes is known as operator overloading.

An operator that works in one way when applied to an instance of a class may work differently on instances of another class. We can also override the overloaded operators in subclasses. For example, we can make comparison operators work in a different way in a superclass and its subclasses.

Taking advantage of polymorphism

We can use the same method, the same name, and same arguments to cause different things to happen according to the class in which we invoke a method. In object-oriented programming, this feature is known as *polymorphism*.

For example, consider that we define a `talk` method in the `Animal` class. The different subclasses of `Animal` must override this method to provide its own implementation of `talk`.

A `Dog` class will override this method to print the representation of a dog barking, that is, a `Woof` message. On the other hand, a `Cat` class will override this method to print the representation of a cat meowing, that is, a `Meow` message.

Now, let's think about a `CartoonDog` class that represents a dog that can really talk as part of a cartoon. The `CartoonDog` class will override the `talk` method to print a `Hello` message because the dog can really talk.

Thus, depending on the type of the instance, we will see a different result after invoking the same method along with the same arguments, even if all of them are subclasses of the same base class, that is, the `Animal` class.

Working with simple inheritance in Python

Firstly, we will create a base class in Python. Then we will use simple inheritance to create subclasses. We will then override methods and overload comparison operators to be able to compare different instances of a specific class and its subclasses. We will take advantage of this polymorphism.

Creating classes that specialize behavior in Python

Now it is time to code all the classes in Python. The following lines show the code for the `Animal` class in Python. The class header doesn't specify a base class; therefore, this class inherits from an object, that is, the most base type for any class in Python. Remember that we are working with Python 3.x and that the syntax to achieve the same goal in Python 2.x is different. In Python 3.x, a class that doesn't specify a base class implicitly inherits from an object:

```
class Animal:
    _number_of_legs = 0
```

```
        _pairs_of_eyes = 0

        def __init__(self, age):
            self._age = age
            print("Animal created")

        @property
        def age(self):
            return self._age

        @age.setter
        def age(self, age):
            self._age = age

        def print_legs_and_eyes(self):
            print("I have " + str(self._number_of_legs) + " legs and " +
    str(self._pairs_of_eyes * 2) + " eyes.")

        def print_age(self):
            print("I am " + str(self._age) + " years old.")
```

The `Animal` class in the preceding code declares two protected class attributes initialized to `0`: `_number_of_legs` and `_pairs_of_eyes`. The `__init__` method requires an `age` value to create an instance of the `Animal` class and prints a message indicating that an animal has been created. This class encapsulates the `_age` protected instance attribute as an `age` property. In addition, the `Animal` class defines the following two instance methods:

- `print_legs_and_eyes`: This method displays the total number of eyes based on the `_pairs_of_eyes` value
- `print_age`: This method displays the age based on the `_age` value

We have to add more code to this class to be able to compare the age of all the different `Animal` instances using operators. We will add the necessary code to this class later.

Using simple inheritance in Python

The following lines show the code for the `Mammal` class that inherits from `Animal`. Note the `class` keyword followed by the class name: `Mammal`, the superclass from which it inherits enclosed in parenthesis `(Animal)`, and a colon (`:`) that contains the header of the class definition:

```
class Mammal(Animal):
    _pairs_of_eyes = 1

    def __init__(self, age, is_pregnant=False):
        super().__init__(age)
        self._is_pregnant = is_pregnant
        print("Mammal created")

    @property
    def is_pregnant(self):
        return self._is_pregnant

    @is_pregnant.setter
    def is_pregnant(self, is_pregnant):
        self._is_pregnant = is_pregnant
```

The `Mammal` class overwrites the value of the `_pairs_of_eyes` protected class attribute with `1`. Remember that the protected class attribute was present in the `Animal` class body, but initialized with `0`.

The `__init__` method requires an `age` value to create an instance of a class and specifies an additional optional argument, `is_pregnant`, whose default value is `False`. If we don't specify the value for `is_pregnant`, Python will use the default value indicated in the method declaration. We cannot declare multiple `__init__` method versions with a different number of parameters within a class in Python; therefore, we can take advantage of optional parameters.

The first line in the `__init__` method invokes the `__init__` method defined in the superclass, that is, the `Animal` class. This superclass defined the `__init__` method with just one argument: the `age` value; therefore, we call it using the `age` value received as an argument (`age`) in our `__init__` method:

```
super().__init__(age)
```

We use `super()` to reference the superclass of the current class.

The `__init__` method defined in the superclass initializes the value for the `_age` protected instance attribute and prints a message indicating that an `Animal` instance has been created. When the `__init__` method returns a value, the following code initializes the value of the `_is_pregnant` instance attribute and prints a message indicating that a mammal has been created:

```
self._is_pregnant = is_pregnant
print("Mammal created")
```

Overriding methods in Python

The following lines show the code for the `DomesticMammal` class that inherits from `Mammal`. The `class` keyword followed by the class name `DomesticMammal`, the superclass from which it inherits enclosed in parenthesis `(Mammal)`, and a colon (`:`) that composes the header of the class definition is shown in the following code:

```
class DomesticMammal(Mammal):
    def __init__(self, name, age, favorite_toy, is_pregnant=False):
        super().__init__(age, is_pregnant)
        self._name = name
        self._favorite_toy = favorite_toy
        print("DomesticMammal created")


    @property
    def name(self):
        return self._name

    @property
    def favorite_toy(self):
        return self._favorite_toy

    @favorite_toy.setter
    def favorite_toy(self, favorite_toy):
        self._favorite_toy = favorite_toy

    def talk(self):
        print(self._name + ": talks")
```

The `__init__` method requires a `name`, an `age`, and a `favorite_toy` to create an instance of a class. In addition, this method specifies an additional optional argument (`is_pregnant`) whose default value is `False`. As in the `Mammal` class, the first line in the `__init__` method invokes the `__init__` method defined in the superclass, that is, the `Mammal` class. The superclass defined the `__init__` method with two arguments: `age` and `is_pregnant`. Thus, we call the `__init__` method using arguments that have the same names in our `__init__` method:

```
super().__init__(age, is_pregnant)
```

After it finishes initializing attributes, the `__init__` method prints a message indicating that a `DomesticMammal` class has been created. This class defines a `name` read-only property and a `favorite_toy` property that encapsulate the `_name` and `_favorite_toy` protected instance attributes. The `talk` instance method displays a message with the `_name` value, followed by a colon (`:`) and `talks`.

The following lines of code show the code for the `Dog` class that inherits from `DomesticMammal`:

```
class Dog(DomesticMammal):
    _number_of_legs = 4
    _breed = "Just a dog"
    _breed_family = "Dog"

    def __init__(self, name, age, favorite_toy, is_pregnant=False):
        super().__init__(name, age, favorite_toy, is_pregnant)
        print("Dog created")

    def bark(self, times=1, other_domestic_mammal=None, is_
angry=False):
        message = self.name
        if other_domestic_mammal is not None:
            message += " to " + other_domestic_mammal.name + ": "
        else:
            message += ": "
        if is_angry:
            message += "Grr "
        message += "Woof " * times
        print(message)

    def talk(self):
        self.bark()

    @classmethod
    def print_breed(cls):
        print(cls._breed)

    @classmethod
    def print_breed_family(cls):
        print(cls._breed_family)
```

The `Dog` class in the preceding code overrides the `talk` method from **`DomesticMammal`** in **`Dog`**. As with the `__init__` method that has been overridden in all the subclasses we have been creating, we just declare this method with the same name. There is no need to add any decorator or keyword in Python to override a method defined in a superclass.

However, in this case the `talk` method doesn't invoke the `__init__` method with the same name for its superclass, that is, we don't use `super()` to invoke the `talk` method defined in `DomesticMammal`. The `talk` method in the `Dog` class invokes the `bark` method without parameters because dogs bark and don't talk.

The **`bark`** method declaration includes three optional arguments. This way, we can call it without parameters or with values for different optional arguments. Python doesn't allow us to overload methods; therefore, we can take advantage of optional arguments. The **`bark`** method prints a message based on the specified number of times (`times`), the destination domestic mammal (`other_domestic_mammal`), and whether the dog is angry or not (`is_angry`).

The `Dog` class also declares two class attributes: `_breed` and `_breed_family`. We will override the values of these attributes in the subclasses of `Dog`. The `print_breed` class method displays the value of the `_breed` class attribute and the `print_breed_family` class method displays the value of the `_breed_family` class attribute.
We won't override these class methods in our subclasses because we just need to override the values of the class attributes to achieve our goals. If we call these class methods from an instance of a subclass of `Dog`, these methods will execute the code specified in the `Dog` class, that is, the last class in the class hierarchy to override the `talk` method, but this code will use the value of the class attributes overridden in subclasses. Thus we will see a message that displays the values of all the class attributes as defined in our subclasses.

The following lines show the code for the `TerrierDog` class that inherits from `Dog`:

```
class TerrierDog(Dog):
    _breed = "Terrier dog"
    _breed_family = "Terrier"

    def __init__(self, name, age, favorite_toy, is_pregnant=False):
        super().__init__(name, age, favorite_toy, is_pregnant)
        print("TerrierDog created")
```

As happened in other subclasses that we have been coding, the `__init__` method requires a `name`, an `age`, and a `favorite_toy` attribute to create an instance of the `TerrierDog` class; we also have the optional `is_pregnant` argument. The `__init__` method invokes a method with the same name defined in the superclass, that is, the `Dog` class. Then the `__init__` method prints a message indicating that a `TerrierDog` instance has been created. The `Dog` class sets `"Terrier dog"` and `"Terrier"` as `_`, which is the value for the `breed` and `_breed_family` class attributes defined in the superclass.

The following lines show the code for the `SmoothFoxTerrier` class that inherits from `TerrierDog`:

```
class SmoothFoxTerrier(TerrierDog):
    _breed = "Smooth Fox Terrier"

    def __init__(self, name, age, favorite_toy, is_pregnant=False):
        super().__init__(name, age, favorite_toy, is_pregnant)
        print("SmoothFoxTerrier created")
```

The `SmoothFoxTerrier` class sets `"Smooth Fox Terrier"` as the value for the `_breed` class attribute defined in the `Dog` class. The `__init__` method invokes a method with the same name defined in the superclass, that is the `TerrierDog` class and then prints a message. This message indicates that a `SmoothFoxTerrier` class instance has been created.

Overloading operators in Python

We want to be able to compare the age of different `Animal` instances using the following operators in Python:

- Less than (`<`)
- Less or equal than (`<=`)
- Greater than (`>`)
- Greater or equal than (`>=`)

We can overload operators in Python to achieve our goals by overriding special instance methods that Python invokes under the hood whenever we use all the operators to compare instances of `Animal`. We have to override the following methods in the **`Animal`** class:

- `__lt__`: This method gets invoked when we use the less than (`<`) operator
- `__le__`: This method gets invoked when we use the less than or equals to (`<=`) operator
- `__gt__`: This method gets invoked when we use the greater than (`>`) operator
- `__ge__`: This method gets invoked when we use the greater than or equals to (`>=`) operator

All the preceding instance methods have the same declaration. Python passes the instance specified at the right-hand side of the operator as an argument, which is usually named as `other`. Thus we have `self` and `other` as the arguments for the instance method, and we must return a `bool` value with the result of the application of the operator, in our case with the result of the comparison operator.

Let's consider that we have two instances of `Animal` or any of its subclasses named `animal1` and `animal2`. If we enter `print(animal1 < animal2)` on the Python console, Python will invoke the `animal1.__lt__` method with `self` equal to `animal1` and `other` equal to `animal2`. Thus we must return a Boolean value indicating that `self.age < other.age`, is equivalent to `animal1.age < animal2.age`.

We must add the following code to the body of the `Animal` class:

```
def __lt__(self, other):
    return self.age < other.age

def __le__(self, other):
    return self.age <= other.age

def __gt__(self, other):
    return self.age > other.age

def __ge__(self, other):
    return self.age >= other.age
```

Understanding polymorphism in Python

After we code all the classes, we will enter the following lines on a Python console:

```
SmoothFoxTerrier.print_breed()
SmoothFoxTerrier.print_breed_family()
```

The following lines show the messages displayed on the Python console after we enter the preceding lines of code:

```
Smooth Fox Terrier
Terrier
```

We coded the `print_breed` and `print_breed_family` class methods in the `Dog` class and didn't override these methods in any of the subclasses. However, we have overridden the values of the class attributes whose content these methods display: `_breed` and `_breed_family`. The former class attribute is overridden in the `SmoothFoxTerrier` class and the latter in the `TerrierDog` class.

We called the class methods from the `SmoothFoxTerrier` class; therefore, these methods took into account the values of the class attributes overridden in the `TerrierDog` and the `SmoothFoxTerrier` classes.

The following code creates an instance of the `SmoothFoxTerrier` class named `tom`:

```
tom = SmoothFoxTerrier("Tom", 5, "Sneakers")
```

The Python console will display the following messages as a result of all the `__init__` methods that have been called and prints a message indicating that an instance of a class has been created. Remember that we have overridden each `__init__` method in all the different classes up to `SmoothFoxTerrier` and its included code to call the `__init__` method of its superclass and display a message:

```
Animal created
Mammal created
DomesticMammal created
Dog created
TerrierDog created
SmoothFoxTerrier created
```

We don't have six different instances; we just have one instance that calls the `__init__` method of six different classes to perform all the necessary initialization in order to create an instance of `SmoothFoxTerrier`. If we execute the following code on the Python console, all of them will display `True` as the result because `tom` is an instance of `Animal`, `Mammal`, `DomesticMammal`, `Dog`, `TerrierDog`, and `SmoothFoxTerrier`:

```
print(isinstance(tom, Animal))
print(isinstance(tom, Mammal))
print(isinstance(tom, DomesticMammal))
print(isinstance(tom, Dog))
print(isinstance(tom, TerrierDog))
print(isinstance(tom, SmoothFoxTerrier))
```

The following code creates two additional instances of `SmoothFoxTerrier` named `pluto` and `goofy`:

```
pluto = SmoothFoxTerrier("Pluto", 6, "Tennis ball")
goofy = SmoothFoxTerrier("Goofy", 8, "Soda bottle")
```

The following code uses all the four operators that we overloaded in the `Animal` class: greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=). Remember that we have overridden special instance methods in the `Animal` class that Python invokes under the hood whenever we use these operators. In this case, we apply these operators on instances of `SmoothFoxTerrier`, and the `Animal` class inherits the overridden special instance methods from the `Animal` base class. These four operators return the results of comparing the age value of all the different instances:

```
print(tom > pluto)
print(tom < pluto)
print(goofy >= tom)
print(tom <= goofy)
```

The following code calls the `bark` method for the instance named `tom` with a different number of arguments. This way, we can take advantage of all the optional arguments in Python. Remember that we coded the `bark` method in the `Dog` class, and the `SmoothFoxTerrier` class inherits the `bark` method from this superclass:

```
tom.bark()
tom.bark(2)
tom.bark(2, pluto)
tom.bark(3, pluto, True)
```

The following code shows the results of calling the `bark` method using different arguments:

```
Tom: Woof
Tom: Woof Woof
Tom to Pluto: Woof Woof
Tom to Pluto: Grr Woof Woof Woof
```

Working with simple inheritance in C#

First, we will create a base class in C#. Then we will use simple inheritance to create subclasses and specialize behavior. We will override methods and overload comparison operators to be able to compare different instances of a specific class and its subclasses. We will take advantage of this polymorphism.

Creating classes that specialize behavior in C#

Now it is time to code all the classes in C#. The following lines show the code for the `Animal` abstract class in C#. The class declaration doesn't specify a base class; therefore, this class inherits from `Object`, specifically `System.Object`. `System.Object` is the base class for all the classes included in .NET Framework. The usage of the `abstract` keyword before `class` makes this class an abstract class that we cannot use to create instances:

```
public abstract class Animal
{
  protected virtual int NumberOfLegs { get { return 0; } }
  protected virtual int PairsOfEyes { get { return 0; } }

  public int Age { get; set; }

  public Animal(int age)
```

```
      {
        this.Age = age;
        Console.WriteLine("Animal created.");
      }

      public void PrintLegsAndEyes()
      {
        Console.WriteLine(
          String.Format("I have {0} legs and {1} eyes.",
          this.NumberOfLegs,
          this.PairsOfEyes * 2));
      }

      public void PrintAge()
      {
        Console.WriteLine(
          String.Format("I am {0} years old."),
          this.Age);
      }
    }
```

The preceding class declares two read-only properties: `NumberOfLegs` and `PairsOfEyes`. Both these properties return `0` as its value. The usage of the `virtual` keyword allows you to override properties in any subclass of `Animal`.

In C#, we have to specify the properties or methods that we allow our subclasses to override by adding the `virtual` keyword. If we don't include the `virtual` keyword, a property or method cannot be overridden and we will see a compiler error if we try to do so.

The constructor requires an `age` value to create an instance of a class and prints a message indicating that an animal has been created. This class uses auto-implemented properties to generate the `Age` property. In addition, the `Animal` class defines the following two instance methods:

- `PrintLegsAndEyes`: This method displays the total number of eyes based on the `PairsOfEyes` value
- `PrintAge`: This method displays the age based on the `age` value

We have to add more code to this class to be able to compare the age of all the different `Animal` instances using operators. We will add the necessary code to this class later.

Using simple inheritance in C#

We will create many classes in C# which require the following `using` statements. We will dive deep into how the `using` statement works and the organization of object-oriented code in *Chapter 7, Organization of Object-Oriented Code*:

```
using System;
using System.Linq;
using System.Text;
```

The following lines show the code for the `Mammal` abstract class that inherits from `Animal`. Note the `class` keyword followed by the `Mammal` class name, a colon (`:`), and `Animal`: the superclass from which it inherits in the class definition:

```
public abstract class Mammal: Animal
{
    protected override int PairsOfEyes { get { return 1; } }
    public bool IsPregnant { get; set; }

    private void Init(bool isPregnant)
    {
        this.IsPregnant = isPregnant;
        Console.WriteLine("Mammal created.");
    }

    public Mammal(int age) : base(age)
    {
        this.Init(false);
    }

    public Mammal(int age, bool isPregnant) : base(age)
    {
        this.Init(isPregnant);
    }
}
```

The `Mammal` class name overrides the `PairsOfEyes` property and defines a new getter method that returns `1`. Remember that the protected class attribute was declared using the `virtual` keyword in the `Animal` class body, but the getter method returned `0`. In this case, the property declaration uses the `override` keyword to override the property declaration of the superclass.

[We will use the `virtual` keyword to indicate that a property or method can be overridden in subclasses. Also, we will use the `override` keyword to override a property or method that was declared with the `virtual` keyword in a superclass.]

Note that this `Animal` class declares two constructors. One of the constructors requires the `age` value to create an instance of a class. The other constructor requires the `age` and `isPregnant` value. If we create an instance of this class with just one `int` argument, C# will use the first constructor. If we create an instance of this class with two arguments: one `int` value and one `bool` value, C# will use the second constructor. Thus, we have overloaded the constructor and provided two different constructors. Of course, we can also take advantage of optional parameters. However, in this case we want to overload constructors.

The lines that declare two constructors are followed by a colon(`:`). A call to the constructor of the superclass with the `age` value is received as an argument. The `base` keyword enables you to call the superclass' constructor. Once the superclass' constructor finishes its execution, both constructors call the `Init` private method which initializes the `IsPregnant` property with a value received as an argument or the default `false` value in case it wasn't specified. The following code shows both constructor declarations:

```
public Mammal(int age) : base(age)
public Mammal(int age, bool isPregnant) : base(age)
```

[We use `base` to reference the superclass' constructor.]

The superclass' constructor initializes the value for the `Age` property and prints a message indicating that an `Animal` instance has been created. When a method returns, the `Init` private method defined in the `Mammal` class initializes the value of the `IsPregnant` property and prints a message indicating that a `Mammal` has been created. Don't forget that we cannot access private methods from subclasses; therefore, the `Init` method is only visible in the `Mammal` class.

Overloading and overriding methods in C#

The following lines show the code for the `DomesticMammal` class that inherits from `Mammal`. Note the `class` keyword followed by the `DomesticMammal` class name, a colon (`:`), and `Mammal`, the superclass from which it inherits in the class definition:

```
public abstract class DomesticMammal: Mammal
{
  public string Name { get; private set; }
  public string FavoriteToy { get; set; }

  private void Init(string name, string favoriteToy)
  {
    this.Name = name;
    this.FavoriteToy = favoriteToy;
    Console.WriteLine("DomesticMammal created.");
  }

  public virtual void Talk()
  {
    Console.WriteLine(String.Format("{0}: talks", this.Name));
  }

  public DomesticMammal(string name, int age, string favoriteToy)
    : base(age)
  {
    this.Init(name, favoriteToy);
  }

  public DomesticMammal(string name, int age, string favoriteToy, bool
isPregnant)
    : base(age, isPregnant)
  {
    this.Init(name, favoriteToy);
  }
}
```

The preceding class declares two constructors. The first constructor requires the `name`, `age`, and `favoriteToy` values to create an instance of a class. The other constructor adds the `isPregnant` argument. As in the `Mammal` class, the lines that declare both constructors are followed by a colon (`:`) and a call to the superclass' constructor. In one case, we just need the `age` value received as an argument, whereas in the other case, it is necessary to add the `isPregnant` value. Once the superclass' constructor finishes its execution, both constructors call the `Init` private method that initializes the properties of `Name` and `FavoriteToy`. After the `Init` method finishes initializing its properties, it prints a message indicating that `DomesticMammal` has been created. The following code shows the declarations of two constructors:

```
public DomesticMammal(string name, int age, string favoriteToy) :
base(age)
public DomesticMammal(string name, int age, string favoriteToy, bool
isPregnant) : base(age, isPregnant)
```

The class defines a `Name` read-only property and a `FavoriteToy` property with autoimplemented properties. The `Talk` instance method displays a message with the `Name` value. This is followed by a colon (`:`) and `talks`. Note that a method uses the `virtual` keyword in its declaration; therefore, we can override it in any subclass.

The following lines show the code for the `Dog` class that inherits from `DomesticMammal`:

```
public class Dog : DomesticMammal
{
  protected override int NumberOfLegs { get { return 4; } }
  public virtual string Breed { get { return "Just a dog"; } }
  public virtual string BreedFamily { get { return "Dog"; } }

  private void Init()
  {
    Console.WriteLine("Dog created.");
  }

  public Dog(string name, int age, string favoriteToy, bool
isPregnant): base(name, age, favoriteToy, isPregnant)
  {
    this.Init();
  }

  public Dog(string name, int age, string favoriteToy)
  : base(name, age, favoriteToy)
  {
    this.Init();
  }

  public void PrintBreed()
  {
    Console.WriteLine(this.Breed);
  }

  public void PrintBreedFamily()
  {
    Console.WriteLine(this.BreedFamily);
  }

  private void PrintBark(int times, DomesticMammal
otherDomesticMammal, bool isAngry)
  {
    var sb = new StringBuilder();
    sb.Append(this.Name);
    if (otherDomesticMammal != null)
    {
      sb.Append(String.Format(" to {0}: ", otherDomesticMammal.Name));
    }
    else
    {
      sb.Append(": ");
    }

    if (isAngry)
    {
      sb.Append("Grr ");
    }
    sb.Append(string.Concat(Enumerable.Repeat("Woof ", times)));
    Console.WriteLine(sb.ToString());
  }

  public void Bark()
  {
    this.PrintBark(1, null, false);
  }

  public void Bark(int times)
  {
    this.PrintBark(times, null, false);
  }

  public void Bark(int times, DomesticMammal otherDomesticMammal)
  {
    this.PrintBark(times, otherDomesticMammal, false);
  }

  public void Bark(int times, DomesticMammal otherDomesticMammal, bool
isAngry)
  {
    this.PrintBark(times, otherDomesticMammal, isAngry);
  }

  public override void Talk()
  {
    this.Bark();
  }
}
```

The `Dog` class overrides the `Talk` method from `DomesticMammal` in `Dog`. As in the overridden properties of other subclasses, we just add the `override` keyword to the method declaration. This method doesn't invoke a method with the same name for its superclass, that is, we don't use the `base` keyword to invoke the `Talk` method defined in `DomesticMammal`. The `Talk` method in the `Dog` class invokes the `Bark` method without parameters because dogs bark and don't talk.

The `Bark` method is overloaded with four declarations with different arguments. The following lines show all the four different declarations included in the class body:

```
public void Bark()
public void Bark(int times)
public void Bark(int times, DomesticMammal otherDomesticMammal)
public void Bark(int times, DomesticMammal otherDomesticMammal, bool
isAngry)
```

This way, we can call any of the defined `Bark` methods based on the arguments that are provided. All the four methods end up invoking the `PrintBark` private method with different default values for all the arguments not provided in the call to `Bark`. The method uses `StringBuilder` to build and print a message based on the specified number of times (`times`), the destination domestic mammal (`otherDomesticMammal`), and whether the dog is angry or not (`isAngry`).

The Dog class also declares two read-only properties: Breed and BreedFamily. We will override the values of these properties in the subclasses of Dog so that they include the virtual keyword in their declaration. The PrintBreed instance method displays the value of the Breed property attribute, whereas the PrintBreedFamily instance method displays the value of the BreedFamily property. We won't override these instance methods in our subclasses because we just need to override the values of two read-only properties to achieve our goals. If we call these instance methods from an instance of a subclass of Dog, these methods will execute the code specified in the Dog class but this code will use the value of all the properties overridden in subclasses. Thus, we will see the messages that displays the values of all the properties as defined in subclasses.

The following lines show the code for the TerrierDog class that inherits from Dog:

```
public class TerrierDog : Dog
{
  public override string Breed { get { return "Terrier dog"; } }
  public override string BreedFamily { get { return "Terrier"; } }

  private void Init()
  {
    Console.WriteLine("TerrierDog created.");
  }

  public TerrierDog(string name, int age, string favoriteToy)
  : base(name, age, favoriteToy)
  {
    this.Init();
  }

  public TerrierDog(string name, int age, string favoriteToy, bool
isPregnant)
  : base(name, age, favoriteToy, isPregnant)
  {
    this.Init();
  }
}
```

As in the other subclasses that we have been coding, we have more than one constructor defined for a class. In this case, one of the constructors require the `name`, `age`, and `favoriteToy` values to create an instance of the `TerrierDog` class, and we also have a constructor that adds the `isPregnant` argument. Both constructors invoke the superclass' constructor and then call the private `Init` method. This method prints a message. This message indicates that `TerrierDog` has been created. The class sets `"Terrier dog"` and `"Terrier"` as the value for the properties of `Breed` and `BreedFamily` that were defined in the superclass and overridden in this `TerrierDog` class.

The following lines show the code for the `SmoothFoxTerrier` class that inherits from `TerrierDog`:

```
public class SmoothFoxTerrier : TerrierDog
{
  public override string Breed { get { return "Smooth Fox Terrier"; }
}

  private void Init()
  {
    Console.WriteLine("Smooth Fox Terrier created.");
  }

  public SmoothFoxTerrier(string name, int age, string favoriteToy)
  : base(name, age, favoriteToy)
  {
    this.Init();
  }

  public SmoothFoxTerrier(string name, int age, string favoriteToy,
bool isPregnant)
  : base(name, age, favoriteToy, isPregnant)
  {
    this.Init();
  }
}
```

The `SmoothFoxTerrier` class sets `"Smooth Fox Terrier"` as the value for the `Breed` property defined in the `Dog` class, which is overridden in this class. The `SmoothFoxTerrier` class defines two constructors with exactly the same parameters that were specified in the two constructors defined in the superclass. Both constructors invoke two constructors defined in the superclass and then call the `Init` private method. This method prints a message indicating that an instance of the `SmoothFoxTerrier` class has been created.

Overloading operators in C#

We want to be able to compare the age of different `Animal` instances using the following operators in C#:

- Less than (`<`)
- Less than or equal to (`<=`)
- Greater than (`>`)
- Greater than or equal to (`>=`)

We can overload the preceding operators in C# to achieve our goals by declaring operators in the `Animal` class that work as static methods that receive two arguments. C# will invoke operators under the hood whenever we use these operators to compare instances of `Animal`. We have to declare the following operators in the **`Animal`** class:

- `<`: This operator is invoked when we use the less than (`<`) operator
- `<=`: This operator is invoked when we use the less than or equal to (`<=`) operator
- `>`: This operator is invoked when we use the greater than (`>`) operator
- `>=`: This operator is invoked when we use the greater than or equal to (`>=`) operator

All the preceding operators have the same declaration. C# passes the instance specified at the left-hand side of the operator as the first argument and the instance specified at the right-hand side of the operator as the second argument. We will use names such as `self` and `other` for these arguments. Thus we have `self` and `other` as the arguments for operators, and we must return a `bool` value with the result of the application of the operator, in our case, with the result of the comparison operator.

Let's consider that we have two instances of `Animal` or any of its subclasses named `animal1` and `animal2`. If we enter `Console.WriteLine(animal1 < animal2);`, C# will invoke the **<** operator for the `Animal` class as a `static` method with `self` equal to `animal1` and `other` equal to `animal2`. Thus we must return a `bool` value indicating that `self.age < other.age` is equivalent to `animal1.age < animal2.age`.

We must add the following code to the body of the `Animal` class:

```
public static bool operator <(Animal self, Animal other)
{
  return self.Age < other.Age;
}

public static bool operator <=(Animal self, Animal other)
{
  return self.Age <= other.Age;
}

public static bool operator >(Animal self, Animal other)
{
  return self.Age > other.Age;
}

public static bool operator >=(Animal self, Animal other)
{
  return self.Age >= other.Age;
}
```

Understanding polymorphism in C#

After we code all the classes, we can write code in the `Main` method of a console application. The following are the first code of the `Main` method that create an instance of the `SmoothFoxTerrier` class named `tom`. Let's use one of its constructors that doesn't require the `isPregnant` argument:

```
var tom = new SmoothFoxTerrier("Tom", 5, "Sneakers");
tom.PrintBreed();
tom.PrintBreedFamily();
```

The following code shows the messages that will be displayed on the Windows console after we enter the preceding code:

```
Animal created.
Mammal created.
DomesticMammal created.
Dog created.
TerrierDog created.
Smooth Fox Terrier created.
Smooth Fox Terrier
Terrier
```

First, the Windows console displays the messages by each constructor that has been called. Remember that each constructor called its base class constructor and printed a message indicating that an instance of the class has been created. We don't have six different instances; we just have one instance that has been calling the chained constructors of six different classes to perform all the necessary initialization to create an instance of `SmoothFoxTerrier`. If we execute the following code in the **Immediate window** of Visual Studio, all of them will return `true` as the result because `tom` is an `Animal`, a `Mammal`, a `DomesticMammal`, a `Dog`, a `TerrierDog`, and a breed of `SmoothFoxTerrier`:

```
tom is Animal
tom is Mammal
tom is DomesticMammal
tom is Dog
tom is TerrierDog
tom is SmoothFoxTerrier
```

We coded the `PrintBreed` and `PrintBreedFamily` methods in the `Dog` class and we didn't override these methods in any of the subclasses. However, we have overridden the properties whose content these methods display: `Breed` and `BreedFamily`. The former property is overridden in the `SmoothFoxTerrier` class and the latter in the `TerrierDog` class.

The following code creates two additional instances of `SmoothFoxTerrier` named `pluto` and `goofy`. In this case, both code use the constructor that receives the `isPregnant` argument:

```
var pluto = new SmoothFoxTerrier("Pluto", 6, "Tennis ball", false);
var goofy = new SmoothFoxTerrier("Goofy", 8, "Soda bottle", false);
```

The following code uses the four operators that we have overloaded in the `Animal` class: greater than (`>`), less than (`<`), greater than or equal to (`>=`), and less than or equal to (`<=`). In this case, we apply these operators on instances of `SmoothFoxTerrier`, and the `Animal` class inherits the operators from the `Animal` base class. The four operators return the results of comparing the `age` value of the different instances:

```
Console.WriteLine(tom > pluto);
Console.WriteLine(tom < pluto);
Console.WriteLine(goofy >= tom);
Console.WriteLine(tom <= goofy);
```

The following code calls the `Bark` method for the `tom` instance with a different number of arguments. This way, we can take advantage of the `Bark` method that we overloaded four times with different arguments in C#. Remember that we coded the four `Bark` methods in the `Dog` class and the `SmoothFoxTerrier` class inherits the overloaded methods from this superclass:

```
tom.Bark();
tom.Bark(2);
tom.Bark(2, pluto);
tom.Bark(3, pluto, true);
```

The following code shows the results of calling the methods with the different arguments:

```
Tom: Woof
Tom: Woof Woof
Tom to Pluto: Woof Woof
Tom to Pluto: Grr Woof Woof Woof
```

Working with the prototype-based inheritance in JavaScript

First, we will create a constructor function in JavaScript and define properties and methods in its prototype. Then, we will take advantage of prototype-based inheritance in order to create objects that specialize the behavior defined in the baseline prototype. We will override methods and properties.

Creating objects that specialize behavior in JavaScript

Now it is time to code objects in JavaScript. The following code defines the `Animal` empty constructor function in JavaScript, followed by the declaration of properties and functions for the `Animal` prototype:

```
function Animal() {}

Animal.prototype.numberOfLegs = 0;
Animal.prototype.pairsOfEyes = 0;
Animal.prototype.age = 0;

Animal.prototype.printLegsAndEyes = function() {
  console.log("I have " + this.numberOfLegs + " legs and " + this.
pairsOfEyes * 2 + " eyes.");
}

Animal.prototype.printAge = function() {
  console.log("I am " + this.age + " years old.");
}
```

In this case, we will use an empty constructor function and then declare all the things that we want to share with the objects that will use `Animal` as its prototype in `Animal.prototype`. The prototype declares three properties initialized with `0` as its value: `age`, `numberOfLegs`, and `pairsOfEyes`.

In addition, the `Animal` prototype defines the following two methods:

- `printLegsAndEyes`: This method displays the total number of eyes based on the `pairsOfEyes` value
- `printAge`: This method displays the age based on the `age` value

We have to add more code to the object's prototype to be able to compare the age of the different `Animal` instances by calling methods. We will add the necessary code to the object's prototype later.

Using the prototype-based inheritance in JavaScript

The following is the code for the empty `Mammal` construction function. Now, let's set the `Mammal.prototype` property to a new instance of the previously defined `Animal` object. This way, we will be able to access the properties and methods defined in the `Animal` object in each `Mammal` instance:

```
function Mammal() {}
Mammal.prototype = new Animal();
Mammal.prototype.constructor = Mammal;
Mammal.prototype.isPregnant = false;
Mammal.prototype.pairsOfEyes = 1;
```

After we change the value of the `Mammal.prototype` property, we will assign the `Mammal` constructor function to the `Mammal.constructor` property in order to clean up the side effects on the constructor property when you change the value of a prototype. We add the `isPregnant` property initialized to `false`. Finally, we overwrite the value of the `pairsOfEyes` property with `1`. Remember that this property was declared in `Animal.prototype`.

Any instance of `Mammal` will be able to access the properties and methods declared in `Animal.prototype`.

Overriding methods in JavaScript

Here is the code for the empty `DomesticMammal` construction function. Here we will set the `DomesticMammal.prototype` property to a new instance of the previously defined `Mammal` object. This way, we will be able to access the properties and methods defined in both the `Mammal` and `Animal` objects in each `DomesticMammal` instance. The prototype chain makes it possible to access all the properties and methods defined in each `prototype` property of the different objects:

```
function DomesticMammal() {}

DomesticMammal.prototype = new Mammal();
DomesticMammal.prototype.constructor = DomesticMammal;
DomesticMammal.prototype.name = "";
DomesticMammal.prototype.favoriteToy = "";

DomesticMammal.prototype.talk = function() {
  console.log(this.name + ": talks");
}
```

We will use an empty constructor function again. Then we will declare all the things that we want to share with the objects that will use `DomesticMammal` as its prototype in `DomesticMammal.prototype`. This prototype declares two properties: `name` and `favoriteToy` initialized with an empty string as their value.

In addition, the preceding prototype defines the `talk` method that displays a message with the `name` value, followed by a colon (`:`) and `talks`. Note that this method will be overridden in the `Dog` object.

The following is the code for the empty `Dog` construction function. Then we set the `Dog.prototype` property to a new instance of the previously defined `DomesticMammal` object. This way, we will be able to access the properties and methods defined in the `DomesticMammal`, `Mammal`, and `Animal` objects in each `Dog` instance. We continue to grow the prototype chain:

```
function Dog() {}
Dog.prototype = new DomesticMammal();
Dog.prototype.constructor = Dog;
Dog.prototype.numberOfLegs = 4;
Dog.prototype.breed = "Just a dog";
Dog.prototype.breedFamily = "Dog";

Dog.prototype.printBreed = function() {
  console.log(this.breed);
}

Dog.prototype.printBreedFamily = function() {
  console.log(this.breedFamily);
}
```

The preceding code overwrites the value of the `numberOfLegs` inherited property with `4`. In addition, the code adds two new properties for `Dog.prototype`: `breed` and `breedFamily`. We will overwrite the values of these properties in the new objects that will have `Dog` as a prototype. The `printBreed` method displays the value of the `breed` property, and the `printBreedFamily` method displays the value of the `breedFamily` property. We won't override these methods in the objects that will have `Dog` as a prototype because we just need to overwrite the values of the properties to achieve our goals. If we call these methods from an instance of an object that includes `Dog` in its prototype chain, these methods will execute the function code declared in the `Dog` prototype, but the code will use the value of the properties overridden in specific objects. Thus, we will see messages that display the values of the properties as defined in the objects that include `Dog` in their prototype chain.

The following code declares a `bark` method and overrides the `talk` method inherited from `DomesticMammal` in `Dog`:

```
Dog.prototype.bark = function(times, otherDomesticMammal, isAngry) {
  var message = this.name;
  if (otherDomesticMammal) {
    message += " to " + otherDomesticMammal.name + ": ";
  }
  else {
    message += ": ";
  }
  if (isAngry) {
    message += "Grr ";
  }
  if (!times) {
    times = 1;
  }
  message += new Array(times + 1).join( "Woof " );
  console.log(message);
}

Dog.prototype.talk = function() {
  this.bark(1);
}
```

The `talk` method overridden in the `Dog` prototype invokes the `bark` method without parameters because dogs bark and don't talk. The `bark` method builds and prints a message according to the specified number of times (`times`), the destination of the domestic mammal (`otherDomesticMammal`), and whether the dog is angry or not (`isAngry`).

The following lines show the code for the `TerrierDog` constructor function and its prototype that inherits from `Dog`:

```
function TerrierDog() { }
TerrierDog.prototype = new Dog();
TerrierDog.prototype.constructor = TerrierDog;
TerrierDog.prototype.breed = "Terrier dog";
TerrierDog.prototype.breedFamily = "Terrier";
```

The `TerrierDog` class sets `"Terrier dog"` and `"Terrier"` as the value for the `breed` and `breedFamily` properties that were defined in `Dog`.

The following lines show the code for the `SmoothFoxTerrier` constructor function and its prototype that inherits from `TerrierDog`:

```
function SmoothFoxTerrier() { }
SmoothFoxTerrier.prototype = new TerrierDog();
SmoothFoxTerrier.prototype.constructor = TerrierDog;
SmoothFoxTerrier.prototype.breed = "Smooth Fox Terrier";

SmoothFoxTerrier.create = function (name, age, favoriteToy,
isPregnant) {
  var dog = new SmoothFoxTerrier();
  dog.name = name;
  dog.age = age;
  dog.favoriteToy = favoriteToy;
  dog.isPregnant = isPregnant;

  return dog;
}
```

The `breed` object sets `"Smooth Fox Terrier"` as the value for the `breed` property defined in the `Dog` object and overridden in this object. In addition, the preceding code declares a `create` function for the `SmoothFoxTerrier` construction function. The `create` function receives `name`, `age`, `favoriteToy`, and `isPregnant` as arguments, creates a new instance of `SmoothFoxTerrier`, and assigns the values received as arguments to the properties with the same name. Finally, the `create` function returns the created and initialized instance of `SmoothFoxTerrier`.

Overloading operators in JavaScript

We want to be able to compare the age of all the different `Animal` instances. JavaScript doesn't allow you to overload operators; therefore, we can create methods to achieve our goal.

We can add the following methods to `Animal.prototype`, which will be available in the prototype chain:

- `lessThan`: Less than (<)
- `lessOrEqualThan`: Less than or equal to (<=)
- `greaterThan`: Greater than (>)
- `greaterOrEqualThan`: Greater than or equal to (>=)

All the preceding methods have the same declaration. They will receive the instance that will be located at the right-hand side of the operator. We will use `other` as the only argument for these methods, and we must return a `bool` value with the result of the application of the operator, in our case, with the result of the comparison operator.

Let's consider that we have two instances of `Animal` or objects in the prototype chain named `animal1` and `animal2`. If we enter `console.log(animal1.lessThan(animal2));`, the method must return a `bool` value indicating that `this.age < other.age` is equivalent to `animal1.age < animal2.age`.

We must add the following code to add all the methods to the `Animal` prototype:

```
Animal.prototype.lessThan = function(other) {
  return this.age < other.age;
}

Animal.prototype.lessOrEqualThan = function(other) {
  return this.age <= other.age;
}

Animal.prototype.greaterThan = function(other) {
  return this.age > other.age;
}

Animal.prototype.greaterOrEqualThan = function(other) {
  return this.age >= other.age;
}
```

Understanding polymorphism in JavaScript

After we code all the constructor functions and fill up their prototypes with properties and methods, we can enter the following code on a JavaScript console. These lines call the `SmoothFoxTerrier.create` method to create an instance of `SmoothFoxTerrier` named `tom` and then call the `printBreed` and `printBreedFamily` methods:

```
var tom = SmoothFoxTerrier.create("Tom", 5, "Sneakers");
tom.printBreed();
tom.printBreedFamily();
```

The following command lines display the messages displayed on the Python console after we enter the previous code:

```
Smooth Fox Terrier
Terrier
```

We coded the `printBreed` and `printBreedFamily` methods in the prototype of the `Dog` constructor function; we didn't override these methods in any of the objects in the prototype chain. However, we have overridden the values of properties whose content these methods display: `breed` and `breedFamily`. The former property is overridden in the `SmoothFoxTerrier` prototype and the latter in the `TerrierDog` prototype.

We called the class methods from the `tom` instance; therefore, these methods took into account the values of the properties overridden in the `TerrierDog` and `SmoothFoxTerrier` objects.

If we execute the following code on the JavaScript console, all of them will display `true` as its result because `tom` is an instance of `Animal`, `Mammal`, `DomesticMammal`, `Dog`, `TerrierDog`, and `SmoothFoxTerrier`:

```
console.log(tom instanceof Animal);
console.log(tom instanceof Mammal);
console.log(tom instanceof DomesticMammal);
console.log(tom instanceof Dog);
console.log(tom instanceof TerrierDog);
console.log(tom instanceof SmoothFoxTerrier);
```

The following code creates two additional instances of `SmoothFoxTerrier`: `pluto` and `goofy`:

```
var pluto = SmoothFoxTerrier.create("Pluto", 6, "Tennis ball");
var goofy = SmoothFoxTerrier.create("Goofy", 8, "Soda bottle");
```

The following code uses the four methods that we declared in the prototype of the `Animal` object: `greaterThan` (>), `lessThan` (<), `greaterOrEqualThan` (>=), and `lessOrEqualThan` (<=). In this case, we invoke the methods on instances of `SmoothFoxTerrier` and the object inherits these methods from the `Animal` object. The methods return the results of comparing the `age` value of the different instances:

```
console.log(tom.greaterThan(pluto));
console.log(tom.lessThan(pluto));
console.log(goofy.greaterOrEqualThan(tom));
console.log(tom.lessOrEqualThan(goofy));
```

The following code calls the `bark` method for the instance named `tom` with a different number of arguments. Remember that we coded the `bark` method in the `Dog` prototype, whereas the `SmoothFoxTerrier` object inherits the method in the prototype chain:

```
tom.bark();
tom.bark(2);
tom.bark(2, pluto);
tom.bark(3, pluto, true);
```

The following code shows the results of calling the methods with all the different arguments:

```
Tom: Woof
Tom: Woof Woof
Tom to Pluto: Woof Woof
Tom to Pluto: Grr Woof Woof Woof
```

Summary

In this chapter, you learned how to take advantage of simple inheritance to specialize a base class. We designed many classes from top to bottom using properties and methods. Then, we coded these classes in Python and C#, taking advantage of the different mechanisms provided by each programming language. We coded different objects and prototypes in JavaScript.

We took advantage of operator overloading in C# and Python. We have overridden methods and properties in subclasses or object prototypes. We took advantage of polymorphism in each programming language.

Now that you learned how to take advantage of inheritance and its related concepts, we are ready to work with multiple inheritance, interfaces, and composition in Python, C#, and JavaScript, which is the topic of the next chapter.

5
Interfaces, Multiple Inheritance, and Composition

In this chapter, we will work with more complex scenarios in which we have to use instances that belong to more than one blueprint. We will use the different features included in each of the three covered programming languages to code an application that requires the combination of multiple blueprints in a single instance. We will:

- Understand how interfaces work in combination with classes
- Work with multiple inheritance of classes in Python
- Take advantage of abstract base classes in Python
- Work with interfaces and multiple inheritance in C#
- Implement interfaces in C#
- Work with composition in JavaScript

Understanding the requirement to work with multiple base classes

We have to work with two different types of characters: comic characters and game characters. A comic character has a nickname and must be able to draw speech balloons and thought balloons. The speech balloon may have another comic character as a destination.

A game character has a full name and must be able to perform the following tasks:

- Draw itself in a specific 2D position indicated by the *x* and *y* coordinates
- Move itself to a specific 2D position indicated by the *x* and *y* coordinates
- Check whether it intersects with another game character

We will work with objects that can be both a comic character and a game character. However, we will also work with objects that are just going to be either a comic character or a game character. Neither the game character nor the comic character has a generic way of performing the previously described tasks. Thus, each object that declares itself as a comic character must define all the tasks related to speech and thought balloons. Each object that declares itself as a game character must define how to draw itself, move, and check whether it intersects with another game character.

An angry dog is a comic character that has a specific way of drawing speech and thought balloons. An angry cat is both a comic character and a game character; therefore, it defines all the tasks required by both character types.

The angry cat is a very versatile character. It can use different costumes to participate in games or comics with different names. An angry cat can also be an alien, a wizard, or a knight.

An alien has a specific number of eyes and must be able to appear and disappear.

A wizard has a spell power score and can make an alien disappear.

A knight has sword power and weight values. He can unsheathe his sword. A common task for the knight is to unsheathe his swords and point it to an alien as a target.

We can create abstract classes to represent a comic character and a game character. Then, each subclass can provide its implementation of the methods. In this case, comic characters and game characters are very different. They don't perform similar tasks that might lead to confusion and problems for multiple inheritance. Thus, we can use multiple inheritance when available to create an angry cat class that inherits from both the comic and game character. In some cases, multiple inheritance is not convenient because similar superclasses might have methods with the same name. Also, it can be extremely confusing to use multiple inheritance.

In addition, we can use multiple inheritance to combine the angry cat class with the alien, wizard, and knight. This way, we will have an angry cat alien, an angry cat wizard, and an angry cat knight. We will be able to use any angry cat alien, angry cat wizard, or angry cat knight as either a comic character or a game character.

Our goals are simple, but we may face a little problem: each programming language provides different features that allow you to code your application. C# doesn't support multiple inheritance of classes, but you can use multiple inheritance with interfaces or combine interfaces with classes. Python supports multiple inheritance of classes, but it doesn't support interfaces. JavaScript doesn't work with classes or interfaces; therefore, it doesn't make sense to try to emulate multiple inheritance in this language. Instead, we will use the best features and the most natural way of each programming language to achieve our goals.

We will use multiple inheritance of classes in Python, and we will also analyze the possibility of working with abstract base classes. We will use interfaces in C# and constructor functions and composition in JavaScript.

Working with multiple inheritance in Python

We will take advantage of multiple inheritance of classes in Python. First, we will declare the base classes that we will use to create other classes that inherit from them. Then, we will create subclasses that inherit from a pair of classes. We will work with instances of these subclasses that inherit from more than one class. Finally, we will analyze the usage of abstract base classes as another way of achieving the same goal with a more strict structure.

Declaring base classes for multiple inheritance

The following lines show the code for the `ComicCharacter` class in Python:

```
class ComicCharacter:
    def __init__(self, nick_name):
        self._nick_name = nick_name

    @property
    def nick_name(self):
        return self._nick_name

    def draw_speech_balloon(self, message, destination):
        pass

    def draw_thought_balloon(self, message):
        pass
```

The preceding class declares a `nick_name` read-only property, a `draw_speech_ballon` method, and a `draw_thought_balloon` method. The `__init__` method receives `nick_name` as an argument and assigns it to the private `_nick_name` attribute that is encapsulated in the `nick_name` property. In fact, the `__init__` method and the property setter are the only two methods that include code. Our subclasses will override the `draw_speech_ballon` and `draw_thought_balloon` methods.

The following lines show the code for the `GameCharacter` class in Python:

```
class GameCharacter:
    def __init__(self, full_name, initial_score, x, y):
        self._full_name = full_name
        self.score = initial_score
        self.x = x
        self.y = y

    @property
    def full_name(self):
        return self._full_name

    def draw(self, x, y):
        pass

    def move(self, x, y):
        pass

    def is_intersecting_with(self, other_character):
        pass
```

In this case, the class declaration includes the `full-name` read-only property and three attributes: `score`, `x`, and `y`. In addition, the class declaration also includes three empty methods: `draw`, `move`, and `is_intersecting_with`. The subclasses of `GameCharacter` will override these methods.

The following lines show the code for the `Alien` class in Python:

```
class Alien:
    def __init__(self, number_of_eyes):
        self.number_of_eyes = number_of_eyes

    def appear(self):
        pass

    def disappear(self):
        pass
```

In this case, the `__init__` method receives a `number_of_eyes` argument and initializes an attribute with the same name. In addition, the class declares two empty methods: `appear` and `disappear`.

The following lines show the code for the `Wizard` class in Python:

```
class Wizard:
    def __init__(self, spell_power):
        self.spell_power = spell_power

    def disappear_alien(self, alien):
        pass
```

In this case, the `__init__` method receives a `spell_power` argument and initializes an attribute with the same name. In addition, the `Wizard` class declares an empty `disappear_alien` method.

The following lines show the code for the `Knight` class in Python:

```
class Knight:
    def __init__(self, sword_power, sword_weight):
        self.sword_power = sword_power
        self.sword_weight = sword_weight

    def unsheath_sword(self, target):
        pass
```

In this case, the `__init__` method receives `sword_power` and `sword_height` as arguments and initializes attributes with the same name. In addition, the `Knight` class declares an empty `unsheath_sword` method.

Declaring classes that override methods

Now, we will declare a class that overrides and implements all the empty methods defined in the `ComicCharacter` class. The following lines show the code for the `AngryDog` class, a subclass of `ComicCharacter`:

```
class AngryDog(ComicCharacter):
    def _speak(self, message):
        print(self.nick_name + ' -> "' + message + '"')

    def _think(self, message):
        print(self.nick_name + ' ***' + message + '***')

    def draw_speech_balloon(self, message, destination):
        if destination is None:
            composed_message = message
        else:
            composed_message = destination.nick_name + ", " + message
        self._speak(composed_message)

    def draw_thought_balloon(self, message):
        self._think(message)
```

The `AngryDog` class doesn't override the `__init__` method; therefore, it uses the method declared in its superclass. Whenever we create an instance of this class, Python will use the `__init__` method defined in the `ComicCharacter` class.

The `AngryDog` class overrides the `draw_speech_balloon` method. This method composes a message based on the value of the `destination` parameter and passes a message to the `_speak` method. This method prints this message in a specific format that includes the `nick_name` value as a prefix. If the `destination` parameter is not equal to `None`, the preceding code uses the value of the `nick_name` property.

In addition, the `AngryDog` class declares the code for the `draw_thought_balloon` method that invokes the `_think` method. This method also prints a message that includes the `nick_name` value as a prefix. So, the `AngryDog` class overrides and implements all the empty methods declared in its superclass, that is, the `ComicCharacter` class.

Now, we will declare another subclass of the `ComicCharacter` class. The following lines show the code for the `AngryCat` class:

```
class AngryCat(ComicCharacter):
    def __init__(self, nick_name, age):
        super().__init__(nick_name)
        self.age = age

    def draw_speech_balloon(self, message, destination):
        if destination is None:
            composed_message = self.nick_name + ' -> "'
            if self.age > 5:
                meow = 'Meow'
            else:
                meow = 'Meeeooow Meeeooow'
            composed_message = '{} -> "{} {}"'.format(self.nick_name,
meow, message)
        else:
            composed_message = '{} === {} ---> {}'.format(
                destination.nick_name,
                self.nick_name,
                message)
        print(composed_message)

    def draw_thought_balloon(self, message):
        print('{} thinks: {}'.format(self.nick_name, message))
```

The `AngryCat` class declares the `__init__` method that overrides the same method declared in the `ComicCharacter` superclass. This method uses `super().__init__` to invoke the `__init__` method of its superclass using `nick_name` as an argument. Then, the preceding code assigns the value of the `age` argument to the `age` attribute.

The `AngryCat` class overrides the `draw_speech_balloon` method. This method composes a message based on the value of the `destination` parameter and the value of the `age` attribute. The `draw_speech_balloon` method prints the generated message. If the `destination` parameter is not equal to `None`, the preceding code uses the value of the `nick_name` property. In addition, the `AngryCat` class declares the code for the `draw_thought_balloon` method.

The `AngryCat` class overrides and implements the empty methods declared in the `ComicCharacter` class. However, this class also declares an additional attribute named `age`.

Declaring a class with multiple base classes

Python allows you to declare a class with multiple base classes or superclasses; therefore, we can inherit attributes, properties, and methods from more than one superclass.

We want the previously coded `AngryCat` class to inherit from the `ComicCharacter` class and the `GameCharacter` class. Thus, we want to use any `AngryCat` instance as a comic character and a game character. In order to do so, we must change the class declaration and add the `GameCharacter` class to the list of superclasses for this class, change the code for the `__init__` method, and override all the empty methods declared in the added superclass.

The following line of code shows the new class declaration. This specifies that the `AngryCat` class inherits from the `ComicCharacter` class and the `GameCharacter` class:

```
class AngryCat(ComicCharacter, GameCharacter):
```

Now, we have to make changes to the `__init__` method because it worked with `super().__init__` to invoke the `__init__` method of its superclass. Now, the `AngryCat` class has two superclasses. It is necessary to call the `__init__` method for both superclasses. In addition, we have to add all the arguments required to call the `__init__` method for the added superclass: `GameCharacter`.

The following code shows the new version of the `__init__` method for the `GameCharacter` class:

```
def __init__(self, nick_name, age, full_name, initial_score, x, y):
    ComicCharacter.__init__(self, nick_name)
    GameCharacter.__init__(self, full_name, initial_score, x, y)
    self.age = age
```

The new `__init__` method receives the `nick_name` argument required to call the `__init__` method of the `ComicCharacter` superclass. Note that we use the `ComicCharacter` class name to call the `__init__` method, and we pass `self` as the first argument. This way, we initialize our instance with the `ComicChracter.__init__` method. Note that we don't use `super()` to call the `__init__` method of the base class because we have two base classes. Also, we need to specify which of them we want to use to call the `__init__` method.

The new `__init__` method also receives the arguments required to call the `__init__` method of the `GameCharacter` superclass: `full_name`, `initial_score`, `x`, and `y`. We use the `GameCharacter` class name to call the `__init__` method for the second superclass, and we pass `self` as the first argument. This way, we initialize our instance using the `GameChracter.__init__` method.

Finally, the new `__init__` method also receives an `age` argument that we use to initialize an attribute with the same name. This way, we invoked the initializers of both superclasses and added our own initialization code.

Now, it is necessary to add the code that overrides all the empty methods defined in the `GameCharacter` class. We have to add the following code to the body of the `GameCharacter` class:

```
def draw(self, x, y):
    self.x = x
    self.y = y
    print('Drawing AngryCat {} at x: {}, y: {}'.format(
        self.full_name,
        str(self.x),
        str(self.y)))

def move(self, x, y):
    self.x = x
    self.y = y
    print('Moving AngryCat {} to x: {}, y: {}'.format(
        self.full_name,
        str(self.x),
        str(self.y)))

def is_intersecting_with(self, other_character):
    return self.x == other_character.x and self.y == other_character.y
```

The `AngryCat` class declares the preceding code to override the `draw`, `move`, and `is_intersecting_with` methods declared in the `GameCharacter` class. This class uses multiple inheritance to make `AngryCat` provide implementations for all the empty methods declared in its two superclasses: `ComicCharacter` and `GameCharacter`.

The following lines show the code for a new `AngryCatAlien` class that inherits from the `AngryCat` class and the `Alien` class:

```
class AngryCatAlien(AngryCat, Alien):
    def __init__(self, nick_name, age, full_name, initial_score, x, y,
number_of_eyes):
        AngryCat.__init__(self, nick_name, age, full_name, initial_
score, x, y)
        Alien.__init__(self, number_of_eyes)

    def appear(self):
        print("I'm {} and you can see my {} eyes.".format(
            self.full_name,
            str(self.number_of_eyes)))

    def disappear(self):
           print('{} disappears.'.format(self.full_name))
```

As a result of the preceding code, we have a new class named `AngryCatAlien`. This is a subclass of the `AngryCat` class and the `Alien` class. The `AngryCatAlien` class is also a subclass of the `ComicCharacter` class and the `GameCharacter` class via the `AngryCat` superclass.

The `__init__` method adds the `number_of_eyes` argument to the argument list defined in the `__init__` method declared in the `AngryCat` superclass. We use the first superclass name called `AngryCat` to call the `__init__` method, and we pass `self` as the first argument. This way, we initialize our instance with the `AngryCat.__init__` method.

Then, we use the second superclass class name called `Alien` to call the `__init__` method, and we pass `self` as the first argument. This way, we initialize our instance with the `Alien.__init__` method. Finally, the `AngryCat` class overrides the empty `appear` and `disappear` methods declared in the `Alien` superclass.

The following lines show the code for a new `AngryCatWizard` class that inherits from the `AngryCat` class and the `Wizard` class:

```
class AngryCatWizard(AngryCat, Wizard):
    def __init__(self, nick_name, age, full_name, initial_score, x, y, 
spell_power):
        AngryCat.__init__(self, nick_name, age, full_name, initial_
score, x, y)
        Wizard.__init__(self, spell_power)

    def disappear_alien(self, alien):
        print('{} uses his {} to make the alien with {} eyes 
disappear.'.format(
            self.full_name,
            self.spell_power,
            alien.number_of_eyes))
```

The `__init__` method adds the `spell_power` argument to the argument list defined in the `__init__` method declared in the `AngryCat` superclass. We use the first superclass name called `AngryCat` to call the `__init__` method, and we pass `self` as the first argument. This way, we initialize our instance with the `AngryCat.__init__` method.

Then, we use the second superclass class name called `Wizard` to call the `__init__` method, and we pass `self` as the first argument. This way, we initialize our instance with the `Wizard.__init__` method. Finally, the `AngryCat` class overrides the empty `disappear_alien` method declared in the `Wizard` superclass.

The `disappear_alien` method receives `alien` as an argument and uses its `number_of_eyes` attribute. Thus, any instance of `AngryCatAlien` would qualify as an argument for this method because it inherits the attribute from the `Alien` superclass.

The following lines show the code for a new `AngryCatKnight` class that inherits from the `AngryCat` class and the `Knight` class:

```
class AngryCatKnight(AngryCat, Knight):
    def __init__(self, nick_name, age, full_name, initial_score, x, y, 
sword_power, sword_weight):
        AngryCat.__init__(self, nick_name, age, full_name, initial_
score, x, y)
        Knight.__init__(self, sword_power, sword_weight)

    def _write_lines_about_the_sword(self):
        print('{} unsheaths his sword.'.format(self.full_name))
        print('Sword Power: {} Sword Weight: {}'.format(
            str(self.sword_power),
            str(self.sword_weight)))

    def unsheath_sword(self, target):
        self._write_lines_about_the_sword()
        if target is not None:
            print('The sword targets an alien with {} eyes.'.format(
                target.number_of_eyes))
```

The `__init__` method adds the `sword_power` and `sword_weight` arguments to the argument list defined in the `__init__` method declared in the `AngryCat` superclass. We use the first superclass name called `AngryCat` to call the `__init__` method, and we pass `self` as the first argument. This way, we initialize our instance with the `AngryCat.__init__` method.

Then, we use the second superclass class name called `Knight` to call the `__init__` method, and we pass `self` as the first argument. This way, we initialize our instance with the `Knight.__init__` method. Finally, the `AngryCatKnight` class overrides the empty `unsheath_sword` method declared in the `Knight` superclass.

The `unsheath_sword` method receives an optional `target` argument as an argument and uses its `number_of_eyes` attribute when it is not equal to `None`. Thus, any instance of `AngryCatAlien` would qualify as an argument for this method because it inherits the attribute from the `Alien` superclass. If the `target` argument is not equal to `None`, the `unsheath_sword` method prints an additional message about the alien that the sword has a target, specifically, the number of eyes.

The following table summarizes all the superclasses for the classes that we have been creating. The superclasses column lists all the superclasses via the inheritance chain:

Class name	**Superclasses**
`AngryDog`	`ComicCharacter`
`AngryCat`	`ComicCharacter` and `GameCharacter`
`AngryCatAlien`	`ComicCharacter`, `GameCharacter`, `AngryCat`, and `Alien`
`AngryCatWizard`	`ComicCharacter`, `GameCharacter`, `AngryCat`, and `Wizard`
`AngryCatKnight`	`ComicCharacter`, `GameCharacter`, `AngryCat`, and `Knight`

Working with instances of classes that use multiple inheritance

Now, we will work with instances of the previously declared classes. The first two lines of the following code creates two instances of the `AngryDog` class named `angry_dog_1` and `angry_dog_2`. Then, this code calls the `draw_speech_balloon` method for `angry_dog_1` twice with a different number of arguments. The second call to this method passes `angry_dog_2` as the second argument because `angry_dog_2` is an instance of `AngryDog`, a class that inherits from the `ComicCharacter` class and includes the `nick_name` property:

```
angry_dog_1 = AngryDog("Brian")
angry_dog_2 = AngryDog("Merlin")

angry_dog_1.draw_speech_balloon("Hello, my name is " + angry_dog_1.
nick_name, None)
angry_dog_1.draw_speech_balloon("How do you do?", angry_dog_2)
angry_dog_2.draw_thought_balloon("Who are you? I think.")
```

The following code creates an instance of the `AngryCat` class named `angry_cat_1`. Its `nick_name` is `Garfield`. The next line calls the `draw_speech_balloon` method for the new instance to introduce Garfield in the comic. Then, `angry_dog_1` calls the `draw_speech_balloon` method and passes `angry_cat_1` as the `destination` argument because `angry_cat_1` is an instance of `AngryCat`, a class that inherits from the `ComicCharacter` class and includes the `nick_name` property. Thus, we can also use instances of `AngryCat` whenever we need an argument that provides either a `nick_name` property or attribute:

```
angry_cat_1 = AngryCat("Garfield", 10, "Mr. Garfield", 0, 10, 20)
angry_cat_1.draw_speech_balloon("Hello, my name is " + angry_cat_1.
nick_name, None)
angry_dog_1.draw_speech_balloon("Hello " + angry_cat_1.nick_name,
angry_cat_1)
```

The following code creates an instance of the `AngryCatAlien` class named `alien_1`. Its `nick_name` is `Alien`. The next line checks whether the call to the `is_intersecting_with` method with the `angry_cat_1` parameter returns `true`. The `is_intersecting_with` method requires an instance that provides the `x` and `y` attributes as the argument. We can use `angry_cat_1` as the argument because one of its superclasses is `ComicCharacter`; therefore, it inherits the attributes of `x` and `y`. The `is_intersecting_with` method will return `true` because the `x` and `y` attributes of both instances have the same value. The line in the `if` block calls the `move` method for `alien_1`. Then, the following code calls the `appear` method:

```
alien_1 = AngryCatAlien("Alien", 120, "Mr. Alien", 0, 10, 20, 3)
```

```
if alien_1.is_intersecting_with(angry_cat_1):
    alien_1.move(angry_cat_1.x + 20, angry_cat_1.y + 20)
alien_1.appear()
```

The following code creates an instance of the `AngryCatWizard` class named `wizard_1`. Its `nick_name` is `Gandalf`. The lines thereafter call the `draw` method and then the `disappear_alien` method with `alien_1` as a parameter. The `draw` method requires an instance that provides the `number_of_eyes` attribute as an argument. We can use `alien_1` as the argument because one of its superclasses is `Alien`; therefore, it inherits the `number_of_eyes` attribute. Then, a call to the `appear` method for `alien1` makes the alien with three eyes appear again:

```
wizard_1 = AngryCatWizard("Gandalf", 75, "Mr. Gandalf", 10000, 30, 40, 
100);
wizard_1.draw(wizard_1.x, wizard_1.y)
wizard_1.disappear_alien(alien_1)
alien_1.appear()
```

The following code creates an instance of the `AngryCatKnight` class named `knight_1`. Its `nick_name` is `Camelot`. The next few lines call the `draw` method and then the `UnsheathSword` method with `alien_1` as a parameter. The `draw` method requires an instance that provides the `number_of_eyes` attribute as the argument. We can use `alien_1` as the argument because one of its superclasses is `Alien`; therefore, it inherits the `number_of_eyes` attribute:

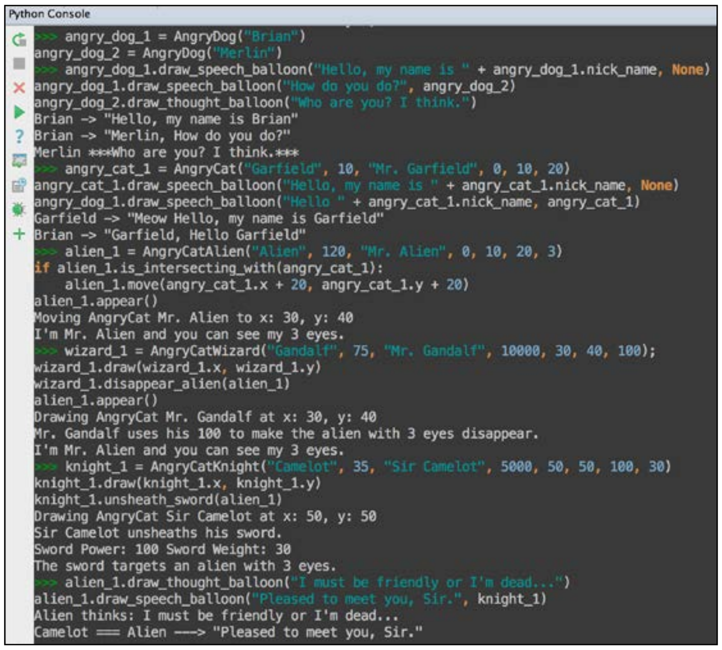
```
knight_1 = AngryCatKnight("Camelot", 35, "Sir Camelot", 5000, 50, 50, 
100, 30)
knight_1.draw(knight_1.x, knight_1.y)
knight_1.unsheath_sword(alien_1)
```

Finally, the following code calls the `draw_thought_balloon` and `draw_speech_balloon` methods for `alien_1`. We can do this because `alien1` is an instance of `AngryCatAlien`; this class inherits all the methods from one of its superclasses: the `AngryCat` class. These methods were declared as empty methods in the `ComicCharacter` class, that is, one of the superclasses of `AngryCat`. The call to the `draw_speech_balloon` method passes `knight_1` as the `destination` argument because `knight_1` is an instance of `AngryCatKnight`. Thus, we can also use instances of `AngryCatKnight` whenever we need an argument that provides either a `nick_name` property or attribute:

```
alien_1.draw_thought_balloon("I must be friendly or I'm dead...")
alien_1.draw_speech_balloon("Pleased to meet you, Sir.", knight_1)
```

After you execute all the preceding code snippets, you will see the following lines on the Python console (see *Figure 1*):

```
Brian -> "Hello, my name is Brian"
Brian -> "Merlin, How do you do?"
Merlin ***Who are you? I think.***
Garfield -> "Meow Hello, my name is Garfield"
Brian -> "Garfield, Hello Garfield"
Moving AngryCat Mr. Alien to x: 30, y: 40
I'm Mr. Alien and you can see my 3 eyes.
Drawing AngryCat Mr. Gandalf at x: 30, y: 40
Mr. Gandalf uses his 100 to make the alien with 3 eyes disappear.
I'm Mr. Alien and you can see my 3 eyes.
Drawing AngryCat Sir Camelot at x: 50, y: 50
Sir Camelot unsheaths his sword.
Sword Power: 100 Sword Weight:30
The sword targets an alien with 3 eyes.
Alien thinks: I must be friendly or I'm dead...
Camelot === Alien ---> "Pleased to meet you, Sir."
```



```
>>> angry_dog_1 = AngryDog("Brian")
angry_dog_2 = AngryDog("Merlin")
>>> angry_dog_1.draw_speech_balloon("Hello, my name is " + angry_dog_1.nick_name, None)
angry_dog_1.draw_speech_balloon("How do you do?", angry_dog_2)
angry_dog_2.draw_thought_balloon("Who are you? I think.")
Brian -> "Hello, my name is Brian"
Brian -> "Merlin, How do you do?"
Merlin ***Who are you? I think.***
>>> angry_cat_1 = AngryCat("Garfield", 10, "Mr. Garfield", 0, 10, 20)
angry_cat_1.draw_speech_balloon("Hello, my name is " + angry_cat_1.nick_name, None)
angry_dog_1.draw_speech_balloon("Hello " + angry_cat_1.nick_name, angry_cat_1)
Garfield -> "Meow Hello, my name is Garfield"
Brian -> "Garfield, Hello Garfield"
>>> alien_1 = AngryCatAlien("Alien", 120, "Mr. Alien", 0, 10, 20, 3)
if alien_1.is_intersecting_with(angry_cat_1):
    alien_1.move(angry_cat_1.x + 20, angry_cat_1.y + 20)
alien_1.appear()
Moving AngryCat Mr. Alien to x: 30, y: 40
I'm Mr. Alien and you can see my 3 eyes.
>>> wizard_1 = AngryCatWizard("Gandalf", 75, "Mr. Gandalf", 10000, 30, 40, 100);
wizard_1.draw(wizard_1.x, wizard_1.y)
wizard_1.disappear_alien(alien_1)
alien_1.appear()
Drawing AngryCat Mr. Gandalf at x: 30, y: 40
Mr. Gandalf uses his 100 to make the alien with 3 eyes disappear.
I'm Mr. Alien and you can see my 3 eyes.
>>> knight_1 = AngryCatKnight("Camelot", 35, "Sir Camelot", 5000, 50, 50, 100, 30)
knight_1.draw(knight_1.x, knight_1.y)
knight_1.unsheath_sword(alien_1)
Drawing AngryCat Sir Camelot at x: 50, y: 50
Sir Camelot unsheaths his sword.
Sword Power: 100 Sword Weight: 30
The sword targets an alien with 3 eyes.
>>> alien_1.draw_thought_balloon("I must be friendly or I'm dead...")
alien_1.draw_speech_balloon("Pleased to meet you, Sir.", knight_1)
Alien thinks: I must be friendly or I'm dead...
Camelot === Alien ---> "Pleased to meet you, Sir."
```

Figure 1

We can use the `isinstance` function with `alien_1`. All the following calls to this function will return `True` because `alien_1` is an instance of the `AngryCatAlien` class and inherits from all its superclasses: `AngryCat`, `Knight`, `ComicCharacter`, and `GameCharacter`:

```
isinstance(alien_1, AngryCat)
isinstance(alien_1, ComicCharacter)
isinstance(alien_1, GameCharacter)
isinstance(alien_1, Alien)
```

Working with abstract base classes

If we want to be stricter and make sure that our classes provide specific methods, the `abc` module (**abstract base classes**) allows you to declare abstract base classes in Python. For example, we can use all the features included in this module to declare the `ComicCharacter` class as an abstract base class in which both `draw_speech_balloon` and `draw_thought_balloon` are abstract methods.

The following lines show the code for a new version of the `ComicCharacter` class, which is declared as an abstract base class:

```
import abc
from abc import ABCMeta
from abc import abstractmethod


class ComicCharacter(metaclass=ABCMeta):
    def __init__(self, nick_name):
        self._nick_name = nick_name

    @abstractmethod
    def draw_speech_balloon(self, message, destination):
        return NotImplemented

    @property
    def nick_name(self):
        return self._nick_name

    @abstractmethod
    def draw_thought_balloon(self, message):
        return NotImplemented
```

The class header specifies `metaclass=ABCMeta`; this is the location in which we specify the superclass. This way, the `abc` module registers a class as an abstract base class and allows you to use specific decorators. Note the usage of the `@absctractmethod` decorator in the `draw_speech_balloon` and `draw_thought_balloon` methods to declare them as abstract methods.

As a result of the declaration of the `ComicCharacter` class with two abstract methods, Python will raise an error whenever we try to create an instance of `ComicCharacter`. For example, we enter the following line in the Python console:

```
scooby = ComicCharacter("Scooby")
```

Python will display the `TypeError: Can't instantiate abstract class ComicCharacter with abstract methods draw_speech_balloon, draw_thought_balloon` error message. This way, we can make sure that we just enable the creation of instances of all the classes that should be instantiated. Without any additional changes to the subclasses of `ComicCharacter`, we can make sure that nobody can create instances of the `ComicCharacter` abstract base class.

Interfaces and multiple inheritance in C#

You can think of an *interface* as a special case of an abstract class. An interface defines properties and methods that a class must implement in order to be considered a member of a group identified with the interface name.

For example, in C#, the language that supports interfaces, we can create the `IAlien` interface that specifies the following elements:

- The `NumberOfEyes` property
- The parameterless method named `Appear`
- The parameterless method named `Disappear`

Once we define an interface, we can use them to specify the required type for an argument. This way, instead of using classes as types, we can use interfaces as types and an instance of any class that implements the specific interface as the argument. For example, if we use `IAlien` as the required type for an argument, we can pass an instance of any class that implements `IAlien` as the argument.

However, you must take into account some limitations of all the interfaces compared with classes. Interfaces cannot declare constructors, destructors, constants, or fields. You cannot specify accessibility modifiers in any members of interfaces. You can declare properties, methods, events, and indexers as members of any interface.

Declaring interfaces

Now, it is time to code all the interfaces in C#. The following lines show the code for the `IComicCharacter` interface in C#. The `public` modifier, followed by the `interface` keyword and the `IComicCharacter` interface name composes the interface declaration. As happens with class declarations, the interface body is enclosed in curly brackets ({}). By convention, interface names start with an uppercase `I` letter:

```
public interface IComicCharacter
{
  string NickName { get; set; }
  void DrawSpeechBalloon(string message);
  void DrawSpeechBalloon(IComicCharacter destination, string message);
  void DrawThoughtBalloon(string message);
}
```

The preceding interface declares the `NickName` string property, the `DrawSpeechBaloon` method overloaded twice, and the `DrawThoughtBalloon` method. The interface includes only the method declaration because all the classes that implement the `IComicCharacter` interface will be responsible for providing the implementation of the two overloads of the `DrawSpeechBalloon` method and the `DrawThoughtBalloon` method. Note that there is no declaration for any constructor.

The following lines show the code for the `IGameCharacter` interface in C#:

```
public interface IGameCharacter
{
  string FullName { get; set; }
  uint Score { get; set; }
  uint X { get; set; }
  uint Y { get; set; }
  void Draw(uint x, uint y);
  void Move(uint x, uint y);
  bool IsIntersectingWith(IGameCharacter otherCharacter);
}
```

In this case, the interface declaration includes four properties: `FullName`, `Score`, `X`, and `Y`. In addition, it also includes three methods: `Draw`, `Move`, and `IsIntersectingWith`. Note that we don't include access modifiers in either the four properties or the three methods.

[We cannot add access modifiers to different members of an interface.]

The following lines show the code for the `IAlien` interface in C#:

```
public interface IAlien
{
  int NumberOfEyes { get; set; }
  void Appear();
  void Disappear();
}
```

In this case, the interface declaration includes the `NumberOfEyes` property and the `Appear` method and the `Disappear` method. Note that we don't include the code for either the getter or setter methods of the `NumberOfEyes` property. As happens with these methods, all the classes that implement the `IAlien` interface will be responsible for providing the implementation of the getter and setter methods for the `NumberOfEyes` property.

The following lines show the code for the `IWizard` interface in C#:

```
public interface IWizard
{
    int SpellPower { get; set; }
    void DisappearAlien(IAlien alien);
}
```

In this case, the interface declaration includes the `SpellPower` property and the `DisappearAlien` method. As happened in other method declarations included in previously declared interfaces, we will use an interface name as the type of an argument in a method declaration. In this case, the `alien` argument for the `DisappearAlien` method is `IAlien`. Thus, we will be able to call the `DisappearAlien` method with any class that implements the `IAlien` interface.

The following lines show the code for the `IKnight` interface in C#:

```
public interface IKnight
{
  int SwordPower { get; set; }
  int SwordWeight { get; set; }
  void UnsheathSword();
  void UnsheathSword(IAlien target);
}
```

In this case, the interface declaration includes two properties: `SwordPower` and `SwordWeight` and the `UnsheathSword` method, which is overloaded twice.

Declaring classes that implement interfaces

Now, we will declare a class that implements the `IComicCharacter` interface. The following lines show the code for the `AngryDog` class. Instead of specifying a superclass, the class declaration includes the name of the previously declared `IComicCharacter` interface after the `AngryDog` class name and the colon (`:`). We can read the class declaration as, "the `AngryDog` class implements the `IComicCharacter` interface":

```
public class AngryDog : IComicCharacter
{
  public string NickName { get; set; }

  public AngryDog(string nickName)
  {
    this.NickName = nickName;
  }

  protected void Speak(string message)
  {
    Console.WriteLine("{0} -> \"{1}\"", this.NickName, message);
  }

  protected void Think(string message)
  {
    Console.WriteLine("{0} -> ***{1}***", this.NickName, message);
  }

  public void DrawSpeechBalloon(string message)
  {
    Speak(message);
  }

  public void DrawSpeechBalloon(IComicCharacter destination, string
message)
  {
    Speak(String.Format("{0}, {1}", destination.NickName, message));
  }

  public void DrawThoughtBalloon(string message)
  {
    Think(message);
  }
}
```

The `AngryDog` class declares a constructor that assigns the value of the required `nickName` argument to the `NickName` property. This class uses auto-implemented properties to declare the `NickName` property and define both the getter and setter methods.

The `AngryDog` class declares the code for the two versions of the `DrawSpeechBalloon` method. Both methods call the protected `Speak` method. This method prints a message on a console in a specific format that includes the `NickName` value as the prefix. In addition, the class declares the code for the `DrawThoughtBalloon` method that invokes the protected `Think` method. This method also prints a message on the console, which includes the `NickName` value as the prefix.

The `AngryDog` class implements the property and all the methods declared in the `IComicCharacter` interface. However, this class also declares two protected members, specifically two protected methods. As long as we implement all the members declared in the interface or interfaces listed in the class declaration, we can add any desired additional member to this class.

Now, we will declare another class that implements the same interface that the `AngryDog` class implemented, that is, the `IComicCharacter` interface. The following lines show the code for the `AngryCat` class:

```
public class AngryCat : IComicCharacter
{
  public string NickName { get; set; }
  public int Age { get; set; }

  public AngryCat(string nickName, int age)
  {
    this.NickName = nickName;
    this.Age = age;
  }

  public void DrawSpeechBalloon(string message)
  {
    if (this.Age > 5)
    {
      Console.WriteLine("{0} -> \"Meow {1}\"", this.NickName, 
message);
    }
    else
    {
      Console.WriteLine("{0} -> \"Meeeooow Meeeooow {1}\"", this.
NickName, message);
    }
  }

  public void DrawSpeechBalloon(IComicCharacter destination, string
message)
  {
    Console.WriteLine("{0} === {1} ---> \"{2}\"", destination.
NickName, this.NickName, message);
  }

  public void DrawThoughtBalloon(string message)
  {
    Console.WriteLine("{0} thinks: {1}", this.NickName, message);
  }
}
```

The `AngryCat` class declares a constructor that assigns the value of the required `nickName` and `age` arguments to the properties of `NickName` and `Age`. This class uses auto-implemented properties to declare the properties of `NickName` and `Age` and their getter and setter methods.

The `AngryCat` class declares the code for the two versions of the `DrawSpeechBalloon` method. The version that requires only a message argument uses the value of the `Age` property to generate a different message when the `Age` value is greater than `5`. In addition, this class declares the code for the `DrawThoughtBalloon` method.

The `AngryCat` class implements the property and all the methods declared in the `IComicCharacter` interface. However, this class also declares an additional property: `Age`, which isn't required by the `IComicCharacter` interface.

If we comment the line that declares the `NickName` property in the `AngryCat` class, the class won't be implementing all the required members of the `IComicCharacter` interface:

```
//public string NickName { get; set; }
```

If we try to compile the code after commenting the previous line, the IDE will display the `Error 1 ConsoleApplication1.AngryCat' does not implement interface member 'ConsoleApplication1.IComicCharacter.NickName'` error. Thus, the compiler enforces you to implement all the members of an interface. If we uncomment the line that declares the `NickName` property, we will be able to compile the project again, as shown in the following code:

```
public string NickName { get; set; }
```

[Interfaces allow you to make sure that all the classes that implement them define all the members specified in the interface. If they don't, the code won't compile.]

Working with multiple inheritance

C# doesn't allow you to declare a class with multiple base classes or superclasses; therefore, there is no support for multiple inheritance of classes. A subclass can inherit from just one class. However, a class can implement one or more interfaces. In addition, we can declare classes that inherit from a superclass and implement one or more interfaces.

We want the `AngryCat` class to implement the `IComicCharacter` and `IGameCharacter` interfaces. Thus, we want to use any `AngryCat` instance as the comic character and the game character. In order to do so, we must change the class declaration, add the `IGameCharacter` interface to the list of interfaces implemented by the `AngryCat` class, and declare all the members included in this interface in the `AngryCat` class.

The following lines show the new class declaration that specifies that the `AngryCat` class implements the `IComicCharacter` interface and the `IGameCharacter` interface:

```
public class AngryCat : IComicCharacter, IGameCharacter
```

If we try to compile a project after changing the class declaration, it won't compile because we didn't implement all the members required by the `IGameCharacter` interface. The IDE will display the following seven errors:

- **Error 1**: `'ConsoleApplication1.AngryCat' does not implement interface member 'ConsoleApplication1.IGameCharacter.IsIntersectingWith(ConsoleApplication1.IGameCharacter)'`
- **Error 2**: `'ConsoleApplication1.AngryCat' does not implement interface member 'ConsoleApplication1.IGameCharacter.Move(uint, uint)'`
- **Error 3**: `'ConsoleApplication1.AngryCat' does not implement interface member 'ConsoleApplication1.IGameCharacter.Draw(uint, uint)'`
- **Error 4**: `'ConsoleApplication1.AngryCat' does not implement interface member 'ConsoleApplication1.IGameCharacter.Y'`
- **Error 5**: `'ConsoleApplication1.AngryCat' does not implement interface member 'ConsoleApplication1.IGameCharacter.X'`

- **Error 6**: `'ConsoleApplication1.AngryCat' does not implement interface member 'ConsoleApplication1.IGameCharacter.Score'`
- **Error 7**: `'ConsoleApplication1.AngryCat' does not implement interface member 'ConsoleApplication1.IGameCharacter.FullName'`

We have to add the following code to the body of the `AngryCat` class in order to implement all the properties specified in the `IGameCharacter` interface with auto-implemented properties:

```
public uint Score { get; set; }
public string FullName { get; set; }
public uint X { get; set; }
public uint Y { get; set; }
```

We have to add the following code to the body of the `AngryCat` class to implement all the methods specified in the `IGameCharacter` interface:

```
public void Draw(uint x, uint y)
{
  X = x;
  Y = y;
  Console.WriteLine("Drawing AngryCat {0} at x: {1}, y: {2}", this.
FullName, x, y);
}

public void Move(uint x, uint y)
{
  X = x;
  Y = y;
  Console.WriteLine("Moving AngryCat {0} to x: {1}, y: {2}", this.
FullName, x, y);
}

public bool IsIntersectingWith(IGameCharacter otherCharacter)
{
  return (this.X == otherCharacter.X) && (this.Y == otherCharacter.Y);
}
```

Now, the `AngryCat` class declares the code for all the three methods: `Draw`, `Move`, and `IsIntersectingWith`. These are required to comply with the `IGameCharacter` interface. Finally, it is necessary to replace the previous constructor with a new one that requires additional arguments and sets the initial values of the recently added properties. The following lines show the code for the new constructor:

```
public AngryCat(string nickName, int age, string fullName, uint
initialScore, uint x, uint y)
```

```
{
    this.NickName = nickName;
    this.Age = age;
    this.FullName = fullName;
    this.Score = initialScore;
    this.X = x;
    this.Y = y;
}
```

The new constructor assigns the value of all the additionally required arguments: `fullName`, `score`, `x`, and `y` to the `FullName`, `InitialScore`, `X`, and `Y` properties. Thus, we will need to specify more arguments whenever we want to create an instance of the `AngryCat` class.

The following lines show the code for a new `AngryCatAlien` class that inherits from the `AngryCat` class and implements the `IAlien` interface. Note that the class declaration includes the `AngryCat` superclass and the implemented `IAlien` interface separated by a comma after the colon (`:`):

```
public class AngryCatAlien : AngryCat, IAlien
{
  public int NumberOfEyes { get; set; }

  public AngryCatAlien(string nickName, int age, string fullName, uint
initialScore, uint x, uint y, int numberOfEyes)
    : base(nickName, age, fullName, initialScore, x, y)
  {
    this.NumberOfEyes = numberOfEyes;
  }

  public void Appear()
  {
    Console.WriteLine("I'm {0} and you can see my {1} eyes.", this.
FullName, this.NumberOfEyes);
  }

  public void Disappear()
  {
    Console.WriteLine("{0} disappears.", this.FullName);
  }
}
```

As a result of the previous code, we have a new class named `AngryCatAlien` that implements the following interfaces:

- `IComicCharacter`: This interface is implemented by the `AngryCat` superclass and inherited by `AngryCatAlien`
- `IGameCharacter`: This interface is implemented by the `AngryCat` superclass and inherited by `AngryCatAlien`
- `IAlien`: This interface is implemented by `AngryCatAlien`

The new constructor adds the `numberOfEyes` argument to the argument list defined in the base constructor, that is, the constructor defined in the `AngryCat` superclass. In this case, the constructor calls the base constructor. Then, it initializes the `NumberOfEyes` property with the value received in the `numberOfEyes` argument. The `AngryCat` class implements the `Appear` and `Disappear` methods required by the `IAlien` interface.

The following lines show the code for the new `AngryCatWizard` class that inherits from the `AngryCat` class and implements the `IWizard` interface. Note that the class declaration includes the `AngryCat` superclass and the implemented `IWizard` interface separated by a comma after the colon (`:`):

```
public class AngryCatWizard : AngryCat, IWizard
{
  public int SpellPower { get; set; }

  public AngryCatWizard(string nickName, int age, string fullName,
uint initialScore, uint x, uint y, int spellPower)
  : base(nickName, age, fullName, initialScore, x, y)
  {
    this.SpellPower = spellPower;
  }

  public void DisappearAlien(IAlien alien)
  {
    Console.WriteLine(
    "{0} uses his {1} spell power to make the alien with {2} eyes
disappear.",
    this.FullName,
    this.SpellPower,
    alien.NumberOfEyes);
  }
}
```

As happened with the `AngryCatAlien` class, the new `AngryCatWizard` class implements three interfaces. Two of these interfaces are implemented by the `AngryCat` superclass and inherited by `AngryCatWizard: IComicCharacter` and `IGameCharacter`. The `AngryCatWizard` class adds the implementation of the `IWizard` interface.

The constructor adds a `spellPower` argument to the argument list defined in the base constructor, that is, the constructor defined in the `AngryCat` superclass. The constructor calls the base constructor and then initializes the `SpellPower` property with the value received in the `spellPower` argument. The `AngryCatWizard` class implements the `DisappearAlien` method required by the `IWizard` interface.

The `DisappearAlien` method receives the `IAlien` interface as the argument. Thus, any instance of `AngryCatAlien` would qualify as the argument for this method, that is, any instance of any class that implements the `IAlien` interface.

The following lines show the code for the new `AngryCatKnight` class that inherits from the `AngryCat` class and implements the `IKnight` interface. Note that the class declaration includes the `AngryCat` superclass and the implemented `IKnight` interface separated by a comma after the colon (`:`):

```
public class AngryCatKnight : AngryCat, IKnight
{
  public int SwordPower { get; set; }
  public int SwordWeight { get; set; }

  public AngryCatKnight(
  string nickName, int age, string fullName,
  uint initialScore, uint x, uint y,
  int swordPower, int swordWeight)
  : base(nickName, age, fullName, initialScore, x, y)
  {
    this.SwordPower = swordPower;
    this.SwordWeight = swordWeight;
  }

  private void WriteLinesAboutTheSword()
  {
    Console.WriteLine(
    "{0} unsheaths his sword.",
    this.FullName);
    Console.WriteLine(
    "Sword power: {0}. Sword Weight: {1}.",
    this.SwordPower,
    this.SwordWeight);
  }

  public void UnsheathSword()
  {
    this.WriteLinesAboutTheSword();
  }

  public void UnsheathSword(IAlien target)
  {
    this.WriteLinesAboutTheSword();
    Console.WriteLine(
    "The sword targets an alien with {0} eyes.",
    target.NumberOfEyes);
  }
}
```

As happened with the two previously coded classes that inherited from the `AngryCat` class and implemented an interface, the new `AngryCatKnight` class implements three interfaces. Two of these interfaces are implemented by the `AngryCat` superclass and inherited by `AngryCatKnight: IComicCharacter` and `IGameCharacter`. The `AngryCatKnight` class adds the implementation of the `IKnight` interface.

The constructor adds the `swordPower` and `swordWeight` arguments to the argument list defined in the base constructor, that is, the constructor defined in the `AngryCat` superclass. This constructor calls the base constructor and then initializes the `SwordPower` and `SwordWeight` properties with the values received in the `swordPower` and `swordHeight` arguments.

The `AngryCat` class implements the two versions of the `UnsheathSword` method required by the `IKnight` interface. Both methods call the private `WriteLinesAboutTheSword` method and the overloaded version that receives the `IAlien` interface as the argument. It prints an additional message about the alien that the sword has a target: the number of eyes.

The following table summarizes all the interfaces implemented by each of the classes that we have been creating:

Class name	**Implemented interfaces**
`AngryDog`	`IComicCharacter`
`AngryCat`	`IComicCharacter` and `IGameCharacter`
`AngryCatAlien`	`IComicCharacter`, `IGameCharacter`, and `IAlien`
`AngryCatWizard`	`IComicCharacter`, `IGameCharacter`, and `IWizard`
`AngryCatKnight`	`IComicCharacter`, `IGameCharacter`, and `IKnight`

Working with methods that receive interfaces as arguments

The following lines show the code for the `Main` method of a console application that uses all the previously declared classes:

```
public static void Main(string[] args)
{
  var angryDog1 = new AngryDog("Brian");
  var angryDog2 = new AngryDog("Merlin");

  angryDog1.DrawSpeechBalloon(String.Format("Hello, my name is {0}",
angryDog1.NickName));
  angryDog1.DrawSpeechBalloon(angryDog2, "How do you do?");
  angryDog2.DrawThoughtBalloon("Who are you? I think.");

  var angryCat1 = new AngryCat("Garfield", 10, "Mr. Garfield", 0, 10,
20);
  angryCat1.DrawSpeechBalloon(String.Format("Hello, my name is {0}",
angryCat1.NickName));
  angryDog1.DrawSpeechBalloon(angryCat1, String.Format("Hello {0}",
angryCat1.NickName));

  var alien1 = new AngryCatAlien("Alien", 120, "Mr. Alien", 0, 10, 20,
3);
  if (alien1.IsIntersectingWith(angryCat1))
  {
    alien1.Move(angryCat1.X + 20, angryCat1.Y + 20);
  }
  alien1.Appear();

  var wizard1 = new AngryCatWizard("Gandalf", 75, "Mr. Gandalf",
10000, 30, 40, 100);
  wizard1.Draw(wizard1.X, wizard1.Y);
  wizard1.DisappearAlien(alien1);

  alien1.Appear();
  var knight1 = new AngryCatKnight("Camelot", 35, "Sir Camelot", 5000,
50, 50, 100, 30);
  knight1.Draw(knight1.X, knight1.Y);
  knight1.UnsheathSword(alien1);

  alien1.DrawThoughtBalloon("I must be friendly or I'm dead...");
  alien1.DrawSpeechBalloon(knight1, "Pleased to meet you, Sir.");
  Console.ReadLine();
}
```

After you execute the previous console application, you will see the following output on the console output:

```
Brian -> "Hello, my name is Brian"
Brian -> "Merlin, How do you do?"
Merlin -> ***Who are you? I think.***
Garfield -> "Meow Hello, my name is Garfield"
Brian -> "Garfield, Hello Garfield"
Moving AngryCat Mr. Alien to x: 30, y: 40
I'm Mr. Alien and you can see my 3 eyes.
Drawing AngryCat Mr. Gandalf at x: 30, y: 40
Mr. Gandalf uses his 100 spell power to make the alien with 3 eyes
disappear.
I'm Mr. Alien and you can see my 3 eyes.
Drawing AngryCat Sir Camelot at x: 50, y: 50
Sir Camelot unsheaths his sword.
Sword power: 100. Sword Weight: 30.
The sword targets an alien with 3 eyes.
Alien thinks: I must be friendly or I'm dead...
Camelot === Alien ---> "Pleased to meet you, Sir."
```

The first two lines create two instances of the `AngryDog` class: `angryDog1` and `angryDog2`. Then, the code calls the two versions of the `DrawSpeechBalloon` method for `angryDog1`. The second call to this method passes `angryDog2` as the `IComicCharacter` argument because `angryDog2` is an instance of `AngryDog`, a class that implements the `IComicCharacter` interface:

```
var angryDog1 = new AngryDog("Brian");
var angryDog2 = new AngryDog("Merlin");
angryDog1.DrawSpeechBalloon(String.Format("Hello, my name is {0}",
angryDog1.NickName));
angryDog1.DrawSpeechBalloon(angryDog2, "How do you do?");
angryDog2.DrawThoughtBalloon("Who are you? I think.");
```

Bear in mind that when we work with interfaces, we use them to specify the argument types instead of using class names. Multiple classes can implement a single interface; therefore, instances of different classes can qualify as an argument of a specific interface.

The first line in the following code creates an instance of the `AngryCat` class called `angryCat1`. Its `NickName` is `Garfield`. The next line calls the `DrawSpeechBalloon` method for the new instance to introduce `Garfield` as a comic character. Then, `angryDog1` calls the `DrawSpeechBalloon` method and passes `angryCat1` as the `IComicCharacter` argument because `angryCat1` is an instance of `AngryCat`, a class that implements the `IComicCharacter` interface. Thus, we can also use instances of `AngryCat` whenever we need the `IComicCharacter` argument:

```
var angryCat1 = new AngryCat("Garfield", 10, "Mr. Garfield", 0, 10, 
20);
angryCat1.DrawSpeechBalloon(String.Format("Hello, my name is {0}", 
angryCat1.NickName));
angryDog1.DrawSpeechBalloon(angryCat1, String.Format("Hello {0}", 
angryCat1.NickName));
```

The first line in the following code creates an instance of the `AngryCatAlien` class named `alien1`. Its `NickName` is `Alien`. The next line checks whether the call to the `IsIntersectingWith` method with `angryCat1` as a parameter returns `true`. The `IsIntersectingWith` method requires the `IComicCharacter` argument; therefore, we can use `angryCat1`. This method will return `true` because the `X` and `Y` properties of both instances have the same value. The line in the `if` block calls the `Move` method for `alien1`. Then, the code calls the `Appear` method:

```
var alien1 = new AngryCatAlien("Alien", 120, "Mr. Alien", 0, 10, 20, 
3);
if (alien1.IsIntersectingWith(angryCat1))
{
    alien1.Move(angryCat1.X + 20, angryCat1.Y + 20);
}
alien1.Appear();
```

The first line in the following code creates an instance of the `AngryCatWizard` class named `wizard1`. Its `NickName` is `Gandalf`. The next line calls the `Draw` method and then the `DisappearAlien` method with `alien1` as the parameter. The `DisappearAlien` method requires the `IAlien` argument; therefore, we can use `alien1`, the previously created instance of `AngryCatAlien` that implements the `IAlien` interface. Then, a call to the `Appear` method for `alien1` makes the alien with three eyes appear again:

```
var wizard1 = new AngryCatWizard("Gandalf", 75, "Mr. Gandalf", 10000, 
30, 40, 100);
wizard1.Draw(wizard1.X, wizard1.Y);
wizard1.DisappearAlien(alien1);
alien1.Appear();
```

The first line in the following code creates an instance of the `AngryCatKnight` class named `knight1`. Its `NickName` is `Camelot`. The next few lines call the `Draw` method and then the `UnsheathSword` method with `alien1` as the parameter. The method requires the `IAlien` argument; therefore, we can use `alien1`, the previously created instance of `AngryCatAlien` that implements the `IAlien` interface:

```
var knight1 = new AngryCatKnight("Camelot", 35, "Sir Camelot", 5000,
50, 50, 100, 30);
knight1.Draw(knight1.X, knight1.Y);
knight1.UnsheathSword(alien1);
```

Finally, the code calls the `DrawThoughtBalloon` and `DrawSpeechBalloon` methods for `alien1`. We can do this because `alien1` is an instance of `AngryCatAlien`; this class inherits the implementation of the `IComicCharacter` interface from its `AngryCat` superclass. The call to the `DrawSpeechBalloon` method passes `knight1` as the `IComicCharacter` argument because `knight1` is an instance of `AngryCatKnight`, a class that also inherits the implementation of the `IComicCharacter` interface from its `AngryCat` superclass. Thus, we can also use instances of `AngryCatKnight` whenever we need the `IComicCharacter` argument.

Working with composition in JavaScript

As previously explained, JavaScript doesn't provide support for interfaces or multiple inheritance. JavaScript allows you to add properties and methods on the fly; therefore, we might create a function that takes advantage of this possibility to emulate multiple inheritance and generate an object that combines two existing objects, a technique known as mix-in.

However, instead of creating functions to create a mix-in, we will create constructor functions and use compositions to access objects within objects. We want to create an application by taking advantage of the feature provided by JavaScript.

Declaring base constructor functions for composition

The following lines show the code for the `ComicCharacter` constructor function in JavaScript:

```
function ComicCharacter(nickName) {
  this.nickName = nickName;
}
```

The constructor function receives the `nickName` argument and uses this value to initialize the `nickName` field.

The following lines show the code for the `GameCharacter` constructor function in JavaScript:

```
function GameCharacter(fullName, initialScore, x, y) {
  this.fullName = fullName;
  this.initialScore = initialScore;
  this.x = x;
  this.y = y;
}
```

The constructor function receives four arguments: `fullName`, `score`, `x`, and `y` and uses these values to initialize fields with the same name.

The following lines show the code for the `Alien` constructor function in JavaScript:

```
function Alien(numberOfEyes) {
  this.numberOfEyes = numberOfEyes;
}
```

The constructor function receives the `numberOfEyes` argument and uses this value to initialize the `numberOfEyes` field.

The following lines show the code for the `Wizard` constructor function in JavaScript:

```
function Wizard(spellPower) {
  this.spellPower = spellPower;
}
```

The constructor function receives the `spellPower` argument and uses this value to initialize the `spellPower` field.

The following lines show the code for the `Knight` constructor function in JavaScript:

```
function Knight(swordPower, swordHeight) {
  this.swordPower = swordPower;
  this.swordHeight = swordHeight;
}
```

The constructor function receives two arguments: `swordPower` and `swordHeight`. The function uses these values to initialize fields with the same name.

We declared five constructor functions that receive arguments and initialize fields with the same name used for all the arguments. We will use these constructor functions to create instances that we will save within fields of other objects.

Declaring constructor functions that use composition

Now, we will declare a constructor function that saves the `ComicCharacter` instance in the `comicCharacter` field. The following lines show the code for the `AngryDog` constructor function:

```
function AngryDog(nickName) {
  this.comicCharacter = new ComicCharacter(nickName);

  Object.defineProperty(this, "nickName", {
    get: function() {
      return this.comicCharacter.nickName;
    }
  });

  this.drawSpeechBalloon = function(message, destination) {
    var composedMessage = "";
    if (destination) {
      composedMessage = destination.nickName + ", " + message;
    } else {
      composedMessage = message;
    }
    console.log(this.nickName + ' -> "' + composedMessage + '"');
  }

  this.drawThoughtBalloon = function(message) {
    console.log(this.nickName + ' ***' + message + '***')
  }
}
```

The `AngryDog` constructor function receives `nickName` as its argument. The function uses this argument to call the `ComicCharacter` constructor function in order to create an instance and save it in the `comicCharacter` field.

The preceding code defines a `nickName` read-only property whose getter function returns the value of the `nickName` field of the previously created `ComicCharacter` object, which is accessed through `this.comicCharacter.nickName`. This way, whenever we create the `AngryDog` object and retrieve the value of its nickname property, the instance will use the saved `ComicCharacter` object to return the value of its `nickName` field.

The `AngryDog` constructor function declares the `drawSpeechBalloon` method. This method composes a message based on the value of the `message` and `destination` parameters and prints a message in a specific format that includes the `nickName` value as its prefix. If the `destination` parameter is specified, the preceding code uses the value of the `nickName` field or property.

In addition, the constructor function declares the code for the `drawThoughtBalloon` method. This method also prints a message with the `nickName` value as its prefix. So, the `AngryDog` constructor function defines two methods that use the `ComicCharacter` object to access its `nickName` field through the `nickName` property.

Now, we will declare another constructor function that saves the `ComicCharacter` instance in the `comicCharacter` field. The following lines show the code for the `AngryCat` constructor function:

```
function AngryCat(nickName, age) {
  this.comicCharacter = new ComicCharacter(nickName);
  this.age = age;

  Object.defineProperty(this, "nickName", {
    get: function() {
      return this.comicCharacter.nickName;
    }
  });

  this.drawSpeechBalloon = function(message, destination) {
    var composedMessage = "";
    if (destination) {
      composedMessage = destination.nickName + ' === ' +
      this.nickName + ' ---> "' + message + '"';
    } else {
      composedMessage = this.nickName + ' -> "';
      if (this.age > 5) {
        composedMessage += "Meow";
      } else {
        composedMessage += "Meeeooow Meeeooow";
      }
      composedMessage += ' ' + message + '"';
    }
    console.log(composedMessage);
  }

  this.drawThoughtBalloon = function(message) {
    console.log(this.comicCharacter.nickName + ' ***' + message +
'***');
  }
}
```

The `AngryCat` constructor function receives two arguments: `nickName` and `age`. This function uses the `nickname` argument to call the `ComicCharacter` constructor function in order to create an instance and save it in the `comicCharacter` field. The code initializes the `age` field with the value received in the `age` argument.

As happened in the `AngryCat` constructor function, the preceding code also defines the `nickName` read-only property whose getter function returns the value of the `nickName` field of the previously created `ComicCharacter` object, which is accessed through `this.comicCharacter.nickName`. This way, whenever we create an `AngryCat` object and retrieve the value of its nickname property, the instance will use the saved `ComicCharacter` object to return the value of its `nickName` field.

The `AngryDog` constructor function declares the `drawSpeechBalloon` method that composes a message based on the value of the `age` attribute and the values of the `message` and `destination` parameters. The `drawSpeechBalloon` method prints a message in a specific format that includes the `nickName` value as its prefix. If the `destination` parameter is specified, the code uses the value of the `nickName` field or property.

In addition, the constructor function declares the code for the `drawThoughtBalloon` method. This method also prints a message including the `nickName` value as its prefix. So, as happened with `AngryDog`, the `AngryCat` constructor function defines two methods that use the `ComicCharacter` object to access its `nickName` field through the `nickName` property.

Working with an object composed of many objects

We want the previously coded `AngryCat` constructor function to be able to work as both the comic and game character. In order to do so, we will add some arguments to the constructor function and save the `GameCharacter` instance in the `gameCharacter` field (among other changes). Here is the code for the new `AngryCat` constructor function:

```
function AngryCat(nickName, age, fullName, initialScore, x, y) {
  this.comicCharacter = new ComicCharacter(nickName);
  this.gameCharacter = new GameCharacter(fullName, initialScore, x,
y);
  this.age = age;
```

We added the necessary arguments: `fullName`, `initialScore`, `x`, and `y` to create the `GameCharacter` object. This way, we have access to the `GameCharacter` object through `this.gameCharacter`. The following code adds the same read-only `nickName` property that we defined in the previous version of the constructor function and four additional properties:

```
Object.defineProperty(this, "nickName", {
  get: function() {
    return this.comicCharacter.nickName;
  }
});

Object.defineProperty(this, "fullName", {
  get: function() {
    return this.gameCharacter.fullName;
  }
});

Object.defineProperty(this, "score", {
  get: function() {
    return this.gameCharacter.score;
  },
  set: function(val) {
    this.gameCharacter.score = val;
  }
});

Object.defineProperty(this, "x", {
  get: function() {
    return this.gameCharacter.x;
  },
  set: function(val) {
    this.gameCharacter.x = val;
  }
});

Object.defineProperty(this, "y", {
  get: function() {
    return this.gameCharacter.y;
  },
  set: function(val) {
    this.gameCharacter.y = val;
  }
});
```

The preceding code defines the `fullName` read-only property whose getter function returns the value of the `fullName` field of the previously created `GameCharacter` object, which is accessed through `this.gameCharacter.fullName`. This way, whenever we create the `AngryCat` object and retrieve the value of its `fullName` property, the instance will use the saved `GameCharacter` object to return the value of its `fullName` field. The other three new properties (`score`, `x`, and `y`) use the same technique with the difference that they also define setter methods that assign a new value to the field with the same name defined in the `GameCharacter` object.

In this case, the `GameCharacter` object uses fields and doesn't define properties with a specific code—such as validations—in the setter method. However, imagine a more complex scenario in which the `GameCharacter` object requires many validations and defines properties instead of fields. We would be reusing these validations by delegating the getter and setter methods to the `GameCharacter` object just by reading and writing to its properties.

The following code defines the methods that we defined in the previous version of the constructor function:

```
this.drawSpeechBalloon = function(message, destination) {
  var composedMessage = "";
  if (destination) {
    composedMessage = destination.nickName + ' === ' +
    this.nickName + ' ---> "' + message + '"';
  } else {
    composedMessage = this.nickName + ' -> "';
    if (this.age > 5) {
      composedMessage += "Meow";
    } else {
      composedMessage += "Meeeooow Meeeooow";
    }
    composedMessage += ' ' + message + '"';
  }
  console.log(composedMessage);
}

this.drawThoughtBalloon = function(message) {
  console.log(this.nickName + ' ***' + message + '***');
}
```

The following code declares three methods: `draw`, `move`, and `isIntersectingWith`. These methods access the previously defined properties of `fullName`, `x`, and `y`:

```
this.draw = function(x, y) {
  this.x = x;
  this.y = y;
  console.log("Drawing AngryCat " + this.fullName +
  " at x: " + this.x +
  ", y: " + this.y);
}

this.move = function(x, y) {
  this.x = x;
  this.y = y;
  console.log("Drawing AngryCat " + this.fullName +
  " at x: " + this.x +
  ", y: " + this.y);
}

this.isIntersectingWith = function(otherCharacter) {
  return ((this.x == otherCharacter.x) &&
  (this.y == otherCharacter.y));
}
```

JavaScript allows you to add attributes, properties, and methods to any object at any time. We take advantage of this feature to extend the object created with the `AngryCat` constructor function to `AngryCat` + `Alien`, `AngryCat` + `Wizard`, and `AngryCat` + `Knight`. We will create and save an instance of the `Alien`, `Wizard`, or `Knight` objects and add the necessary methods and properties to extend our `AngryCat` object on the fly.

The following code declares the `createAlien` method that receives the `numberOfEyes` argument. The method calls the `Alien` constructor function with the `numberOfEyes` value received in the argument and saves the `Alien` instance in the `alien` field. The next few lines add the `numberOfEyes` property. This works as a bridge to the `this.alien.numberOfEyes` attribute. The following code also adds two methods: `appear` and `disappear`:

```
this.createAlien = function(numberOfEyes) {
  this.alien = new Alien(numberOfEyes);

  Object.defineProperty(this, "numberOfEyes", {
    get: function() {
      return this.alien.numberOfEyes;
```

```
        },
        set: function(val) {
          this.alien.numberOfEyes = val;
        }
      });

      this.appear = function() {
        console.log("I'm " + this.fullName +
        " and you can see my " + this.numberOfEyes +
        " eyes.");
      }

      this.disappear = function() {
        console.log(this.fullName + " disappears.");
      }
    }
```

The following code declares the `createWizard` method that receives the `spellPower` argument. The method calls the `Wizard` constructor function with the `spellPower` value received in the argument and saves the `Wizard` instance in the `wizard` field. The next few lines add the `spellPower` property. This works as a bridge to the `this.wizard.spellPower` attribute. The following code also adds the `disappearAlien` method that receives the `alien` argument and uses its `numberOfEyes` field:

```
    this.createWizard = function(spellPower) {
      this.wizard = new Wizard(spellPower);

      Object.defineProperty(this, "spellPower", {
        get: function() {
          return this.wizard.spellPower;
        },
        set: function(val) {
          this.wizard.spellPower = val;
        }
      });

      this.disappearAlien = function(alien) {
        console.log(this.fullName + " uses his " +
        this.spellPower + " to make the alien with " +
        alien.numberOfEyes + " eyes disappear.");
      }
    }
```

Finally, the following code declares the `createKnight` method. This method receives two arguments: `swordPower` and `swordHeight` and calls the `Knight` constructor function with the `swordPower` and `swordHeight` values received in all the arguments and saves the `Knight` instance in the `knight` field. The next few lines add the `swordPower` and `swordHeight` properties that work as a bridge to the `this.wizard.swordPower` and `this.wizard.swordHeight` attributes. The following code also adds the `unsheathSword` method. This method receives the `target` argument and uses its `numberOfEyes` field and calls another new method: `writeLinesAboutTheSword`. Note that with the following lines, we finish the code for the `AngryCat` constructor function:

```
  this.createKnight = function(swordPower, swordHeight) {
    this.knight = new Knight(swordPower, swordHeight);

    Object.defineProperty(this, "swordPower", {
      get: function() {
        return this.knight.swordPower;
      },
      set: function(val) {
        this.knight.swordPower = val;
      }
    });

    Object.defineProperty(this, "swordHeight", {
      get: function() {
        return this.knight.swordHeight;
      },
      set: function(val) {
        this.knight.swordHeight = val;
      }
    });

    this.writeLinesAboutTheSword = function() {
      console.log(this.fullName + " unsheaths his sword.");
      console.log("Sword Power: " + this.swordPower +
      ". Sword Weight: " + this.swordWeight);
    };

    this.unsheathSword = function(target) {
      this.writeLinesAboutTheSword();
      if (target) {
        console.log("The sword targets an alien with " +
        target.numberOfEyes + " eyes.");
      }
    }
  }
}
```

Now, we will code three new constructor functions: `AngryCatAlien`, `AngryCatWizard`, and `AngryCatKnight`. These constructor functions allows you to easily create instances of `AngryCat` + `Alien`, `AngryCat` + `Wizard`, and `AngryCat` + `Knight`.

The following lines show the code for the `AngryCatAlien` constructor function, which receives all the necessary arguments to call the `AngryCat` constructor function. Then, it calls the `createAlien` method for the created object. Finally, the following code returns the object after the call to `createAlien` that added properties and methods:

```
var AngryCatAlien = function(nickName, age, fullName, initialScore, x,
y, numberOfEyes) {
  var alien = new AngryCat(nickName, age, fullName, initialScore, x,
y);
  alien.createAlien(numberOfEyes);
  return alien;
}
```

The following lines show the code for the `AngryCatWizard` constructor function, which receives all the necessary arguments to call the `AngryCat` constructor function. Then, it calls the `createWizard` method for the created object. Finally, the following code returns the object after the call to `createWizard` that added properties and methods:

```
var AngryCatWizard = function(nickName, age, fullName, initialScore,
x, y, spellPower) {
  var wizard = new AngryCat(nickName, age, fullName, initialScore, x,
y);
  wizard.createWizard(spellPower);
  return wizard;
}
```

The following lines show the code for the `AngryCatKnight` constructor function. This function receives all the necessary arguments to call the `AngryCat` constructor function. Then, it calls the `createKnight` method for the created object. Finally, the following code returns the object after the call to `createKnight` that added properties and methods:

```
var AngryCatKnight = function(nickName, age, fullName, initialScore,
x, y, swordPower, swordHeight) {
  var knight = new AngryCat(nickName, age, fullName, initialScore, x,
y);
  knight.createKnight(swordPower, swordHeight);
  return knight;
}
```

The following table summarizes the objects that are included in other objects after we create instances with all the different constructor functions:

Constructor function	**Includes instances of**
`AngryDog`	`ComicCharacter`
`AngryCat`	`ComicCharacter` and `GameCharacter`
`AngryCatAlien`	`AngryCat`, `ComicCharacter`, `GameCharacter`, and `Alien`
`AngryCatWizard`	`AngryCat`, `ComicCharacter`, `GameCharacter`, and `Wizard`
`AngryCatKnight`	`AngryCat`, `ComicCharacter`, `GameCharacter`, and `Knight`

Working with instances composed of many objects

Now, we will work with instances created using all the previously declared constructor functions. In the following code, the first two lines create two `AngryDog` objects named `angryDog1` and `angryDog2`. Then, the code calls the `drawSpeechBalloon` method for `angryDog1` twice with a different number of arguments. The second call to this method passes `angryDog2` as the second argument because `angryDog2` is an `AngryDog` object and includes the `nickName` property:

```
var angryDog1 = new AngryDog("Brian");
var angryDog2 = new AngryDog("Merlin");

angryDog1.drawSpeechBalloon("Hello, my name is " + angryDog1.
nickName);
angryDog1.drawSpeechBalloon("How do you do?", angryDog2);
angryDog2.drawThoughtBalloon("Who are you? I think.");
```

The following code creates the `AngryCat` object named `angryCat1`. Its `nickName` is `Garfield`. The next line calls the `drawSpeechBalloon` method for the new instance to introduce Garfield in the comic character. Then, `angryDog1` calls the `drawSpeechBalloon` method and passes `angryCat1` as the `destination` argument because `angryCat1` is the `AngryCat` object and includes the `nickName` property. Thus, we can also use `AngryCat` objects whenever we need the argument that provides the `nickName` property or field:

```
var angryCat1 = new AngryCat("Garfield", 10, "Mr. Garfield", 0, 10,
20);
angryCat1.drawSpeechBalloon("Hello, my name is " + angryCat1.
nickName);
angryDog1.drawSpeechBalloon("Hello " + angryCat1.NickName, angryCat1);
```

The following code creates the `AngryCatAlien` object named `alien1`. Its `nickName` is `Alien`. The next few lines check whether the call to the `isIntersectingWith` method with `angryCat1` as its parameter returns `true`. The method requires an instance that provides the `x` and `y` fields or properties as the argument. We can use `angryCat1` as the argument because one of its included objects is `ComicCharacter`; therefore, it provides the `x` and `y` attributes. This method will return `true` because the `x` and `y` properties of both instances have the same value. The line within the `if` block calls the `move` method for `alien1`. Then, the following code also calls the `appear` method:

```
var alien1 = AngryCatAlien("Alien", 120, "Mr. Alien", 0, 10, 20, 3);
if (alien1.isIntersectingWith(angryCat1)) {
  alien1.move(angryCat1.x + 20, angryCat1.y + 20);
}
alien1.appear();
```

The first line in the following code creates the `AngryCatWizard` object named `wizard1`. Its `nickName` is `Gandalf`. The next lines call the `draw` method and then the `disappearAlien` method with `alien1` as the parameter. The method requires an instance that provides the `numberOfEyes` field or property as the argument. We can use `alien1` as the argument because one of its included objects is `Alien`; therefore, it includes the `numberOfEyes` field or property. Then, a call to the `appear` method for `alien1` makes the alien with three eyes appear again:

```
var wizard1 = new AngryCatWizard("Gandalf", 75, "Mr. Gandalf", 10000,
30, 40, 100);
wizard1.draw(wizard1.x, wizard1.y);
wizard1.disappearAlien(alien1);
alien1.appear();
```

The first line in the following code creates the `AngryCatKnight` object named `knight1`. Its `nickName` is `Camelot`. The next lines call the `draw` method and then the `unsheathSword` method with `alien1` as the parameter. The method requires an instance that provides the `numberOfEyes` field or property as the argument. We can use `alien1` as the argument because one of its included objects is `Alien`; therefore, it includes the `numberOfEyes` attribute:

```
var knight1 = new AngryCatKnight("Camelot", 35, "Sir Camelot", 5000,
50, 50, 100, 30);
knight1.draw(knight1.x, knight1.y);
knight1.unsheathSword(alien1);
```

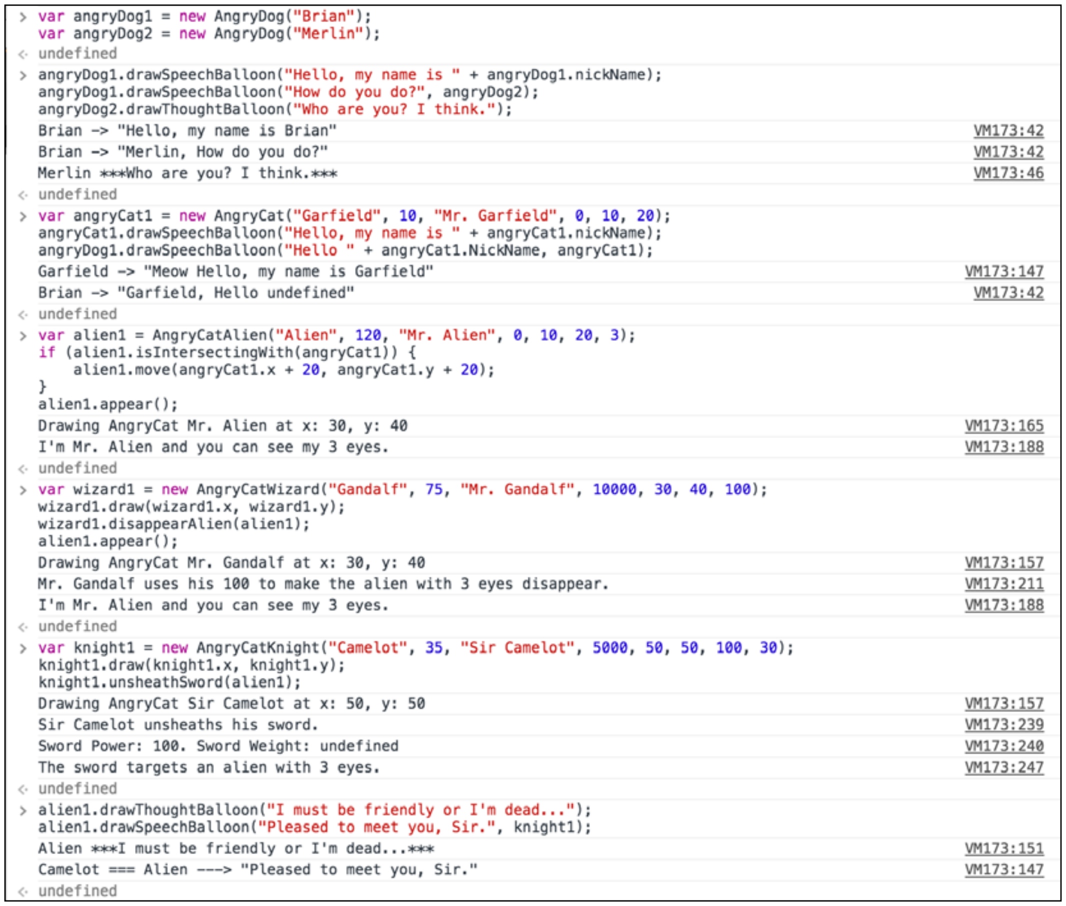
Finally, the following code calls the `drawThoughtBalloon` and `drawSpeechBalloon` methods for `alien1`. We can do this because `alien1` is the `AngryCatAlien` object and includes the methods defined in the `AngryCat` constructor function. The call to the `drawSpeechBalloon` method passes `knight1` as the `destination` argument because `knight1` is the `AngryCatKnight` object. Thus, we can also use instances of `AngryCatKnight` whenever we need an argument that provides the `nickName` property or field:

```
alien1.drawThoughtBalloon("I must be friendly or I'm dead...");
alien1.drawSpeechBalloon("Pleased to meet you, Sir.", knight1);
```

After you execute all the preceding code snippets, you will see the following output on the JavaScript console (see *Figure 2*):

```
Brian -> "Hello, my name is Brian"
Brian -> "Merlin, How do you do?"
Merlin ***Who are you? I think.***
Garfield -> "Meow Hello, my name is Garfield"
Brian -> "Garfield, Hello undefined"
Drawing AngryCat Mr. Alien at x: 30, y: 40
I'm Mr. Alien and you can see my 3 eyes.
Drawing AngryCat Mr. Gandalf at x: 30, y: 40
Mr. Gandalf uses his 100 to make the alien with 3 eyes disappear.
I'm Mr. Alien and you can see my 3 eyes.
Drawing AngryCat Sir Camelot at x: 50, y: 50
Sir Camelot unsheaths his sword.
Sword Power: 100. Sword Weight: undefined
```

```
The sword targets an alien with 3 eyes.
Alien ***I must be friendly or I'm dead...***
Camelot === Alien ---> "Pleased to meet you, Sir."
```



```
> var angryDog1 = new AngryDog("Brian");
  var angryDog2 = new AngryDog("Merlin");
<· undefined
> angryDog1.drawSpeechBalloon("Hello, my name is " + angryDog1.nickName);
  angryDog1.drawSpeechBalloon("How do you do?", angryDog2);
  angryDog2.drawThoughtBalloon("Who are you? I think.");
  Brian -> "Hello, my name is Brian"                                      VM173:42
  Brian -> "Merlin, How do you do?"                                       VM173:42
  Merlin ***Who are you? I think.***                                      VM173:46
<· undefined
> var angryCat1 = new AngryCat("Garfield", 10, "Mr. Garfield", 0, 10, 20);
  angryCat1.drawSpeechBalloon("Hello, my name is " + angryCat1.nickName);
  angryDog1.drawSpeechBalloon("Hello " + angryCat1.NickName, angryCat1);
  Garfield -> "Meow Hello, my name is Garfield"                          VM173:147
  Brian -> "Garfield, Hello undefined"                                    VM173:42
<· undefined
> var alien1 = AngryCatAlien("Alien", 120, "Mr. Alien", 0, 10, 20, 3);
  if (alien1.isIntersectingWith(angryCat1)) {
      alien1.move(angryCat1.x + 20, angryCat1.y + 20);
  }
  alien1.appear();
  Drawing AngryCat Mr. Alien at x: 30, y: 40                             VM173:165
  I'm Mr. Alien and you can see my 3 eyes.                               VM173:188
<· undefined
> var wizard1 = new AngryCatWizard("Gandalf", 75, "Mr. Gandalf", 10000, 30, 40, 100);
  wizard1.draw(wizard1.x, wizard1.y);
  wizard1.disappearAlien(alien1);
  alien1.appear();
  Drawing AngryCat Mr. Gandalf at x: 30, y: 40                           VM173:157
  Mr. Gandalf uses his 100 to make the alien with 3 eyes disappear.      VM173:211
  I'm Mr. Alien and you can see my 3 eyes.                               VM173:188
<· undefined
> var knight1 = new AngryCatKnight("Camelot", 35, "Sir Camelot", 5000, 50, 50, 100, 30);
  knight1.draw(knight1.x, knight1.y);
  knight1.unsheathSword(alien1);
  Drawing AngryCat Sir Camelot at x: 50, y: 50                           VM173:157
  Sir Camelot unsheaths his sword.                                       VM173:239
  Sword Power: 100. Sword Weight: undefined                              VM173:240
  The sword targets an alien with 3 eyes.                                VM173:247
<· undefined
> alien1.drawThoughtBalloon("I must be friendly or I'm dead...");
  alien1.drawSpeechBalloon("Pleased to meet you, Sir.", knight1);
  Alien ***I must be friendly or I'm dead...***                          VM173:151
  Camelot === Alien ---> "Pleased to meet you, Sir."                     VM173:147
<· undefined
```

Figure 2

Summary

In this chapter, you learned how to declare and combine multiple blueprints to generate a single instance. We worked with multiple inheritance of classes in Python. You also learned how to transform a base class to an abstract base class. We declared interfaces in C#. Then, we implemented them with different classes. We also combined interfaces with classes to take advantage of multiple inheritance in C#.

We took advantage of the flexibility of JavaScript that allowed us to add properties and methods to an existing object. We combined constructor functions with composition to generate all the necessary blueprints for our application without creating complex functions that emulate multiple inheritance in a language that wasn't designed with this feature in mind.

Now that we have learned about interfaces, multiple inheritance and composition, we are ready to work with duck typing and generics, which is the topic of the next chapter.

6
Duck Typing and Generics

In this chapter, we will write code that we will maximize code reuse by writing code capable of working with objects of different types. We will take advantage of the different mechanisms to maximize code reuse in each of the three covered programming languages: Python, JavaScript, and C#. We will cover the following topics :

- Understanding parametric polymorphism and generics
- Understanding duck typing
- Working with duck typing in Python
- Working with generics in C#
- Declaring classes that work with one and two constrained generic types in C#
- Working with duck typing in JavaScript

Understanding parametric polymorphism and duck typing

Let's imagine that we want to organize a party of specific animals. We don't want to mix cats with dogs because the party would end up with dogs chasing cats. We want a party, and we don't want intruders. However, at the same time, we want to take advantage of all the procedures we create to organize the party and replicate them with frogs in another party, a party of frogs. We want to reuse these procedures and use them for either dogs or frogs. However, in the future, we will probably want to use them with parrots, lions, tigers, and horses.

In C#, we can declare an interface to specify all the requirements for an animal and write generic code that works with any class that implements the interface. Parametric polymorphism allows you to write generic and reusable code that can work with values without depending on the type while keeping the full static type safety. We can take advantage of parametric polymorphism through generics, also known as generic programming. Once we declare an interface that specifies the requirements for an animal, we can create a class that can work with any instance that implements this interface. This way, we can reuse the code that can generate a party of dogs and create a party of frogs, a party of parrots, or a party of any other animal.

Python's default philosophy is a bit different. Python uses duck typing. Here, the presence of certain attributes or properties and methods make an object suitable to its usage as a specific animal. If we require animals to have a name property and provide sing and dance methods, we can consider any object as an animal as long as it provides the required name property, the sing method, and the dance method. Any instance that provides the required property and methods can be used as an animal.

Let's think about a situation where we see a bird. The bird quacks, swims, and walks like a duck; we can call this bird a duck. Very similar examples related to a bird and a duck generated the duck typing name. We don't need additional information to work with this bird as a duck.

We can add code to constrain types in Python. However, we don't want to write code against Python's most common practices; therefore, we will take advantage of duck typing in Python. We will also take advantage of duck typing in JavaScript. In fact, you might notice that we have been working with duck typing in the examples used in the previous chapters. It is important to note that you can also work with duck typing in C#. However, it requires some workarounds.

Working with duck typing in Python

We will use the `Animal` base class to generalize the requirements for animals. First, we will specialize the base class in two subclasses: `Dog` and `Frog`. Then, we will create a `Party` class that will be able to work with instances of any `Animal` subclass through duck typing. We will work with a party of dogs and a party of frogs.

Then, we will create a `HorseDeeJay` class and generate a subclass of the `Party` class named `PartyWithDeeJay`. The new subclass will work with instances of any `Animal` subclass and any instance that provides the properties and methods declared in the `HorseDeeJay` class through duck typing. We will work with the party of dogs with a DJ.

Declaring a base class that defines the generic behavior

We will create many classes that require the following `import` statement:

```
import random
```

Now, we will declare a base class named `Animal`:

```
class Animal:
    dance_characters = ""
    spelled_sound_1 = ""
    spelled_sound_2 = ""
    spelled_sound_3 = ""

    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

    def dance(self):
        print('{} dances {}'.format(self._name, type(self).dance_
characters))


    def say(self, message):
        print('{} says: {}'.format(self._name, message))

    def say_goodbye(self, destination):
        print('{} says goodbye to {}: {} {} {} '.format(
            self._name,
            destination.name,
            type(self).spelled_sound_1,
            type(self).spelled_sound_2,
            type(self).spelled_sound_1))

    def say_welcome(self, destination):
        print('{} welcomes {}: {}'.format(
            self._name,
            destination.name,
            type(self).spelled_sound_2))


    def sing(self):
        spelled_sing_sound = type(self).spelled_sound_1 + \
            " "
        print('{} sings: {}. {}. {}. '.format(
            self._name,
            spelled_sing_sound * 3,
            spelled_sing_sound * 2,
            spelled_sing_sound))
```

The `Animal` class declares the following four class attributes, all of them initialized with an empty string. The subclasses of `Animal` will override these class attributes with the appropriate strings according to the animal:

- `dance_characters`
- `spelled_sound_1`
- `spelled_sound_2`
- `spelled_sound_3`

Then, the `Animal` class declares an `__init__` method that assigns the value of the required `name` argument to the `_name` protected attribute. This class declares the `name` read-only property that encapsulates the `_name` protected attribute.

The `dance` method uses the value retrieved from the `dance_characters` class attribute to print a message. This message indicates that the animal is dancing. Note the usage of `type(self)` to access the class attribute in a generic way instead of using the actual class name. The `say` method prints the message received as an argument.

Both the `say_welcome` and `say_goodbye` methods receive a `destination` argument that they will use to print the name of the destination of the message. Therefore, whenever we call a method, the destination argument must be an object that has either a name attribute or property in order to be considered as an animal. Any instance of any subclass of `Animal` qualifies as the destination argument for both methods.

The `say_welcome` method uses a combination of strings retrieved from the `spelled_sound_1` and `spelled_sound_3` class attributes to say welcome to another animal. The `say_goodbye` method uses the string retrieved from the `spelled_sound_2` class attribute to say goodbye to another animal.

Declaring subclasses for duck typing

Now, we will create a subclass of `Animal`, a `Dog` class that overrides the string class attributes defined in the `Animal` class to provide all the values that are appropriate for a dog:

```
class Dog(Animal):
    dance_characters = "/-\ \-\ /-/"
    spelled_sound_1 = "Woof"
    spelled_sound_2 = "Wooooof"
    spelled_sound_3 = "Grr"
```

With just a few additional lines of code, we will create another subclass of `Animal`, a `Frog` class that also overrides the string class attributes defined in the `Animal` class to provide all the values that are appropriate for a frog:

```
class Frog(Animal):
    dance_characters = "/|\ \|/ ^ ^ "
    spelled_sound_1 = "Ribbit"
    spelled_sound_2 = "Croak"
    spelled_sound_3 = "Croooaaak"
```

Declaring a class that works with duck typing

The following code declares the `Party` class that takes advantage of duck typing to work with instances of any class that provides either a `name` attribute or property. It implements the `dance`, `say`, `say_goodbye`, `say_welcome`, and `sing` methods. The `__init__` method receives `leader` that the code assigns to the `_leader` protected attribute. In addition, the following code creates a list with `leader` as one of its members and saves it in the `_members` protected attribute. This way, the `leader` argument specifies the first party leader and also the first member of the party, that is, the first element added to the `_members` list:

```
class Party:
    def __init__(self, leader):
        self._leader = leader
        self._members = [leader]
```

The following code declares the `add_member` method that receives the `member` argument. The code adds the member received as an argument to the `_members` list and calls the `_leader.say_welcome` method with `member` as the argument to make the party leader welcome the new member:

```
    def add_member(self, member):
        self._members.append(member)
        self._leader.say_welcome(member)
```

The following code declares the remove_member method that receives the member argument. It checks whether the member to be removed is the party leader. The remove_member method raises a ValueError exception if the member is the party leader. If the member isn't the party leader, the code removes this member from the _members list and calls the say_goodbye method for the removed member. This way, the member who leaves the party says goodbye to the party leader:

```
    def remove_member(self, member):
        if member == self._leader:
            raise ValueError(
                "You cannot remove the leader from the party")
        self._members.remove(member)
        member.say_goodbye(self._leader)
```

The following code declares the dance method that calls the method with the same name for each member of the _members list:

```
    def dance(self):
        for member in self._members:
            member.dance()
```

The following code declares the sing method that calls the method with the same name for each member of the _members list:

```
    def sing(self):
        for member in self._members:
            member.sing()
```

Finally, the following code declares the vote_leader method. This code makes sure that there are at least two members in the _members list when we call this method; if we have just one member, the method raises the ValueError exception. If we have at least two members, the code generates a new random leader (who is different from the existing leader) for the party. The code calls the say method for the actual leader to explain to other party members that another leader has been voted for. Finally, the code calls the dance method for the new leader and sets the new value for the _leader protected attribute:

```
    def vote_leader(self):
        if len(self._members) == 1:
            raise ValueError("You need at least two members to vote a
new Leader.")
        new_leader = self._leader
        while new_leader == self._leader:
            random_leader = random.randrange(len(self._members))
            new_leader = self._members[random_leader]
            self._leader.say('{} has been voted as our new party
leader.'.format(new_leader.name))
        new_leader.dance()
        self._leader = new_leader
```

Using a generic class for multiple types

We have two classes that inherit from the `Animal` class: `Dog` and `Frog`. Both classes have all the required attributes and methods that allow you to work with their instances as arguments of the methods of the previously coded `Party` class. We can start working with instances of the `Dog` class to create a party of dogs.

The following code creates four instances of the `Dog` class: `jake`, `duke`, `lady`, and `dakota`:

```
jake = Dog("Jake")
duke = Dog("Duke")
lady = Dog("Lady")
dakota = Dog("Dakota")
```

The following line of code creates a `Party` instance named `dogsParty` and passes `jake` as the argument. This way, we create the party of frogs in which Jake is the party leader:

```
dogs_party = Party(jake)
```

The following code adds the previously created three instances of `Dog` to the dogs' party by calling the `add_member` method:

```
dogs_party.add_member(duke)
dogs_party.add_member(lady)
dogs_party.add_member(dakota)
```

The following code calls the `dance` method to make all the dogs dance, removes a member who isn't the party leader, votes for a new leader, and finally calls the `sing` method to make all the dogs sing:

```
dogs_party.dance()
dogs_party.remove_member(duke)
dogs_party.vote_leader()
dogs_party.sing()
```

The following lines display the output generated on the Python console after running the preceding code:

```
Jake welcomes Duke: Wooooof
Jake welcomes Lady: Wooooof
Jake welcomes Dakota: Wooooof
Jake dances /-\ \-\ /-/
Duke dances /-\ \-\ /-/
Lady dances /-\ \-\ /-/
Dakota dances /-\ \-\ /-/
Duke says goodbye to Jake: Woof Wooooof Grr
Jake says: Dakota has been voted as our new party leader.
Dakota dances /-\ \-\ /-/
Jake sings: Woof Woof Woof . Woof Woof . Woof .
Lady sings: Woof Woof Woof . Woof Woof . Woof .
Dakota sings: Woof Woof Woof . Woof Woof . Woof .
```

What about the party of frogs? The following code creates four instances of the `Frog` class: `frog1`, `frog2`, `frog3`, and `frog4`:

```
frog1 = Frog("Frog #1")
frog2 = Frog("Frog #2")
frog3 = Frog("Frog #3")
frog4 = Frog("Frog #4")
```

The following code creates a `Party` instance named `frogsParty` and passes `frog1` as the argument. This way, we create the party of dogs in which `Frog #1` is the party leader:

```
frogs_party = Party(frog1)
```

The following code adds the previously created three instances of `Frog` to the frogs' party by calling the `add_member` method:

```
frogs_party.add_member(frog2)
frogs_party.add_member(frog3)
frogs_party.add_member(frog4)
```

The following code calls the `dance` method to make all the frogs dance, removes a member who isn't the party leader, votes for a new leader, and finally calls the `sing` method to make all the frogs sing:

```
frogs_party.dance()
frogs_party.remove_member(frog3)
frogs_party.vote_leader()
frogs_party.sing()
```

The following lines display the output generated in the Python console after running the preceding code:

```
Frog #1 welcomes Frog #2: Croak
Frog #1 welcomes Frog #3: Croak
Frog #1 welcomes Frog #4: Croak
Frog #1 dances /|\ \|/ ^ ^
Frog #2 dances /|\ \|/ ^ ^
Frog #3 dances /|\ \|/ ^ ^
Frog #4 dances /|\ \|/ ^ ^
Frog #3 says goodbye to Frog #1: Ribbit Croak Croooaaak
Frog #1 says: Frog #2 has been voted as our new party leader.
Frog #2 dances /|\ \|/ ^ ^
Frog #1 sings: Ribbit Ribbit Ribbit . Ribbit Ribbit . Ribbit .
Frog #2 sings: Ribbit Ribbit Ribbit . Ribbit Ribbit . Ribbit .
Frog #4 sings: Ribbit Ribbit Ribbit . Ribbit Ribbit . Ribbit .
```

Working with duck typing in mind

Now, we will create a new class that declares properties and methods that we will call from the subclass of the previously created `Party` class. As long as we use instances that provide the required properties and methods, we can use the instances of any class with the new subclass of `Party`. Here is the code for the `HorseDeeJay` class:

```
class HorseDeeJay:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

    def play_music_to_dance(self):
           print("My name is {}. Let's Dance.".format(self.name))

    def play_music_to_sing(self):
        print("Time to sing!")
```

The `HorseDeeJay` class declares the `__init__` method that assigns the value of the required `name` argument to the `_name` protected field. The class declares the `Name` read-only property that encapsulates the related field.

The play_music_to_dance method prints a message that displays the horse DJ name and invites all its party members to dance. The play_music_to_sing method prints a message that invites all its party members to sing.

The following code declares the subclass of the previously created Party class. This class can work with the instance of the HorseDeeJay class for the dee_jay argument required by the __init__ method. Note that the __init__ method receives two arguments: leader and deejay. Both these arguments specify the first party leader, the first member of the party, and the DJ who will make all party members dance and sing. Note that the method calls the __init__ method defined in the Party superclass with leader as an argument:

```
class PartyWithDeeJay(Party):
    def __init__(self, leader, dee_jay):
        super().__init__(leader)
        self._dee_jay = dee_jay
```

The following code declares the dee_jay read-only property that encapsulates the previously created _dee_jay attribute:

```
    @property
    def dee_jay(self):
        return self._dee_jay
```

The following code declares the dance method that overrides the method with the same declaration included in the superclass. The code calls the _dee_jay.play_music_to_dance method. Then, it calls the super().dance method, that is, the dance method defined in the Party superclass:

```
    def dance(self):
        self._dee_jay.play_music_to_dance()
        super().dance()
```

Finally, the following code declares the sing method that overrides the method with the same declaration included in the superclass. The code calls the _dee_jay.play_music_to_sing method and then calls the super().sing method, that is, the sing method defined in the Party superclass:

```
    def sing(self):
        self._dee_jay.play_music_to_sing()
        super().sing()
```

Here is the code that we can run on a Python console to create a `HorseDeeJay` instance named `silver`. Then, the code creates a `PartyWithDeeJay` instance named `silverParty` and passes `jake` and `silver` as arguments. This way, we can create a party with a dog leader and a horse DJ, where Jake is the party leader and Silver is the DJ:

```
silver = HorseDeeJay("Silver")
silverParty = PartyWithDeeJay(jake, silver)
```

The following code adds the previously created three instances of `Dog` to the party by calling the `add_member` method:

```
silverParty.add_member(duke)
silverParty.add_member(lady)
silverParty.add_member(dakota)
```

The following code calls the `dance` method to make the DJ invite all the dogs to dance and then make them dance. Then, the code removes a member who isn't the party leader, votes for a new leader, and finally calls the `sing` method to make the DJ invite all the dogs to sing and then make them sing:

```
silverParty.dance()
silverParty.remove_member(duke)
silverParty.vote_leader()
silverParty.sing()
```

The following lines display the console output after we run the added code:

```
My name is Silver. Let's Dance.
Jake dances /-\ \-\ /-/
Duke dances /-\ \-\ /-/
Lady dances /-\ \-\ /-/
Dakota dances /-\ \-\ /-/
Duke says goodbye to Jake: Woof Wooooof Grr
Jake says: Dakota has been voted as our new party leader.
Dakota dances /-\ \-\ /-/
Time to sing!
Jake sings: Woof Woof Woof . Woof Woof . Woof .
Lady sings: Woof Woof Woof . Woof Woof . Woof .
Dakota sings: Woof Woof Woof . Woof Woof . Woof .
```

Working with generics in C#

We will create an `IAnimal` interface to specify the requirements that a type must meet in order to be considered an animal. We will create the `Animal` abstract base class that implements this interface. Then, we will specialize this class in two subclasses: `Dog` and `Frog`. Later, we will create the `Party` class that will be able to work with instances of any class that implements the `IAnimal` interface through generics. We will work with the party of dogs and frogs.

Now, we will create an **`IDeeJay`** interface and implement it in a `HorseDeeJay` class. We will also create a subclass of the `Party` class named `PartyWithDeeJay` that will use generics to work with instances of any type that implement the `IAnimal` interface and instances of any type that implements the `IDeeJay` interface. Then, we will work with the party of dogs with a DJ.

Declaring an interface to be used as a constraint

Now, it is time to code one of the interfaces that will be used as a constraint later when we define the class that takes advantage of generics. The following lines show the code for the `IAnimal` interface in C#. The `public` modifier, followed by the `interface` keyword and the `IAnimal` interface name composes the interface declaration. Don't forget that we cannot declare constructors within interfaces:

```
public interface IAnimal
{
  string Name { get; set; }

  void Dance();
  void Say(string message);
  void SayGoodbye(IAnimal destination);
  void SayWelcome(IAnimal destination);
  void Sing();
}
```

The `IAnimal` interface declares a `Name` string property and five methods: `Dance`, `Say`, `SayGoodbye`, `SayWelcome`, and `Sing`. The interface includes only the method declaration because the classes that implement the `IAnimal` interface will be responsible for providing the implementation of the getter method and the setter method for the `Name` property and the other five methods.

Declaring an abstract base class that implements two interfaces

Now, we will declare an abstract class named `Animal` that implements both the previously defined `IAnimal` interface and the `IEquatable<IAnimal>` interface. The `IEquatable<T>` interface defines a generalized `Equals` method that we must implement in our class to determine equality of instances. As we are implementing the `IAnimal` interface, we have to replace `T` with `IAnimal` and implement `IEquatable<IAnimal>`. This way, we will be able to determine equality of instances of classes that implement the `IAnimal` interface. We can read the class declaration as "the `Animal` class implements both the `IAnimal` and `IEquatable<Animal>` interfaces":

```
public abstract class Animal: IAnimal, IEquatable<IAnimal>
{
  protected string _name;

  public string Name
  {
    get { return this._name; }
    set { throw new InvalidOperationException("Name is a read-only
property."); }
  }

  public virtual string DanceCharacters { get { return string.Empty; }
}
  public virtual string SpelledSound1 { get { return string.Empty; } }
  public virtual string SpelledSound2 { get { return string.Empty; } }
  public virtual string SpelledSound3 { get { return string.Empty; } }

  public Animal(string name)
  {
    this._name = name;
  }

  public void Dance()
  {
    Console.WriteLine(
    String.Format(
    "{0} dances {1}",
    this.Name,
    DanceCharacters));
  }

  public bool Equals(IAnimal otherAnimal)
```

```
    {
      return (this == otherAnimal);
    }

    public void Say(string message)
    {
      Console.WriteLine(
      String.Format(
      "{0} says: {1}",
      this.Name, message));
    }

    public void SayGoodbye(IAnimal destination)
    {
      Console.WriteLine(
      String.Format(
      "{0} says goodbye to {1}: {2} {3} {4}",
      this.Name,
      destination.Name,
      SpelledSound1,
      SpelledSound3,
      SpelledSound1));
    }

    public void SayWelcome(IAnimal destination)
    {
      Console.WriteLine(
      String.Format(
      "{0} welcomes {1}: {2}",
      this.Name,
      destination.Name,
      SpelledSound2));
    }

    public void Sing()
    {
      var spelledSingSound = SpelledSound1 + " ";
      var sb = new StringBuilder();
      sb.Append(String.Format("{0} sings: ", this.Name));
      sb.Append(String.Concat(Enumerable.Repeat(spelledSingSound, 3)));
      sb.Append(". ");
      sb.Append(String.Concat(Enumerable.Repeat(spelledSingSound, 2)));
      sb.Append(". ");
      sb.Append(spelledSingSound);
```

```
        sb.Append(". ");
        Console.WriteLine(sb.ToString());
      }
    }
```

The `Animal` class declares a constructor that assigns the value of the required `name` argument to the `_name` protected field. This class declares the `Name` read-only property that encapsulates the `_name` private field. The interface requires a `Name` property; therefore, it is necessary to create both setter and getter public methods. We cannot use auto-implemented properties with a private setter because the setter method must be public. Thus, we defined the public setter method that throws an `InvalidOperationException` to avoid users of subclasses of this abstract class to change the value of the `Name` property.

Then, the abstract class declared the following four virtual string properties. All of them define a getter method that returns an empty string that the subclasses will override with the appropriate strings according to the animal:

- `DanceCharacters`
- `SpelledSound1`
- `SpelledSound2`
- `SpelledSound2`

The `Dance` method uses the value retrieved from the `DanceCharacters` property to print a message. This message indicates that the animal is dancing. The `Say` method prints the message received as the argument. Both the `SayWelcome` and `SayGoodbye` methods receive `IAnimal` as the argument that they use to print the name of the destination of the message. `SayWelcome` uses a combination of strings retrieved from `SpelledSound1` and `SpelledSound3` to say welcome to another animal. `SayGoodbye` uses the string retrieved from `SpelledSound2` to say goodbye to another animal.

The `Equals` method receives another `IAnimal` as the argument and uses the `==` operator between the current instance and the received instance to check whether the instances are the same or not. In a more complex scenario, you might want to code this method to compare the values of certain properties to determine equality. In our case, we want to keep the code as simple as possible to focus on generics. We needed to implement the `Equals` method to conform to the `IEquatable<IAnimal>` interface.

Declaring subclasses of an abstract base class

We have the abstract `Animal` class that implements `IAnimal` and `IEquatable<IAnimal>`. Now, we will create a subclass of `Animal`, a `Dog` class that overrides the virtual string properties defined in the `Animal` class to provide the appropriate values for a dog, and declare a constructor that just calls the base constructor:

```
public class Dog: Animal
{
  public override string SpelledSound1
  {
    get { return "Woof"; }
  }
  public override string SpelledSound2
  {
    get { return "Wooooof"; }
  }

  public override string SpelledSound3
  {
    get { return "Grr"; }
  }

  public override string DanceCharacters
  {
    get { return @"/-\ \-\ /-/"; }
  }

  public Dog(string name): base(name)
  {
  }
}
```

With just a few additional lines of code, we will create another subclass of `Animal`, a `Frog` class that also overrides all the virtual string properties defined in the `Animal` class to provide the appropriate values for a frog, and declare a constructor that just calls the base constructor:

```
public class Frog: Animal
{
  public override string SpelledSound1
  {
    get { return "Ribbit"; }
  }

  public override string SpelledSound2
  {
    get { return "Croak"; }
  }

  public override string SpelledSound3
  {
    get { return "Croooaaak"; }
  }

  public override string DanceCharacters
  {
    get { return @"/|\ \|/ ^ ^ "; }
  }

  public Frog(string name)
  : base(name)
  {
  }
}
```

Declaring a class that works with a constrained generic type

The following line declares a `Party` class that takes advantage of generics to work with many types. The class name is followed by a less than sign (`<`), `T` that identifies the generic type parameter, and a greater than sign (`>`). The `where` keyword, followed by `T` that identifies the type and a colon (`:`) indicates that the `T` generic type parameter has to be a type that implements the specified interface, that is, the `IAnimal` interface:

```
  public class Party<T> where T: IAnimal
```

The following line starts the class body and declares a private `List` (`System.Generics.Collection.List`) of the type specified by `T`. `List`, which uses generics to specify the type of all the elements that will be added to the list:

```
  {
    private List<T> _members;
```

The following line declares a public `Leader` property whose type is `T`:

```
    public T Leader { get; private set; }
```

The following code declares the constructor that receives the `leader` argument whose type is `T`. This argument specifies the first party leader and the first member of the party as well, that is, the first element added to the `_members` list:

```
public Party(T leader)
{
  this.Leader = leader;
  this._members = new List<T>();
  this._members.Add(leader);
}
```

The following code declares the `AddMember` method that receives the `member` argument whose type is `T`. The code adds the member received as an argument to the `_members` `List<T>` and calls the `Leader.SayWelcome` method with `member` as the argument to make the party leader welcome the new member:

```
public void AddMember(T member)
{
  this._members.Add(member);
  Leader.SayWelcome(member);
}
```

The following code declares the `RemoveMember` method that receives the `member` argument whose type is `T`. The code checks whether the member to be removed is the party leader. The method throws an exception if the member is the party leader. The code returns the `bool` result, calls the remove method of the `_members` `List<T>` with the member received as an argument, and calls the `SayGoodbye` method for the successfully removed member. This way, the member that leaves the party says goodbye to the party leader:

```
public bool RemoveMember(T member)
{
  if (member.Equals(this.Leader))
  {
    throw new InvalidOperationException("You cannot remove the leader
from the party.");
  }
  var result = this._members.Remove(member);
  if (result)
```

```
    {
      member.SayGoodbye(this.Leader);
    }
    return result;
  }
```

The following code declares the `Dance` method that calls the method with the same name for each member of the `_members` list. We will use the `virtual` keyword because we will override this method in a future subclass:

```
public virtual void Dance()
{
  foreach (var member in _members)
  {
    member.Dance();
  }
}
```

The following code declares the `Sing` method that calls the method with the same name for each member of the `_members` list. We will use the `virtual` keyword because we will override this method in a future subclass:

```
public virtual void Sing()
{
  foreach (var member in _members)
  {
    member.Sing();
  }
}
```

Finally, the following code declares the `VoteLeader` method. This code makes sure that there are at least two members in the `_members` list when we call this method; if we have just one member, the method throws an `InvalidOperationException`. If we have at least two members, the code generates a new random leader (who is different from the existing leader) for the party. The code calls the `Say` method for the actual leader to explain it to all the other party members that another leader has been voted for. Finally, the code calls the `Dance` method for the new leader and sets the new value for the `Leader` property:

```
public void VoteLeader()
{
  if (this._members.Count == 1)
  {
    throw new InvalidOperationException("You need at least two members
to vote a new Leader.");
```

```
    }

    var newLeader = this.Leader;
    while (newLeader.Equals(this.Leader))
    {
      var randomLeader =
      new Random().Next(this._members.Count);
      newLeader = this._members[randomLeader];
    }

    this.Leader.Say(
    String.Format(
    "{0} has been voted as our new party leader.",
    newLeader.Name));
    newLeader.Dance();
    this.Leader = newLeader;
  }
  }
```

Using a generic class for multiple types

We can create instances of the `Party<T>` class by replacing the `T` generic type parameter with any type name that conforms to all the constraints specified in the declaration of the `Party<T>` class. So far, we have two concrete classes: `Dog` and `Frog` that implement the `IAnimal` interface. Thus, we can use `Dog` to create an instance of `Party<Dog>`.

The following code shows the first few lines of the console application that creates four instances of the `Dog` class: `jake`, `duke`, `lady`, and `dakota`. Then, the `Main` method creates the `Party<Dog>` instance named `dogsParty` and passes `jake` as the argument. This way, we create the party of dogs in which Jake is the party leader:

```
  class Program
  {
    static void Main(string[] args)
    {
      var jake = new Dog("Jake");
      var duke = new Dog("Duke");
      var lady = new Dog("Lady");
      var dakota = new Dog("Dakota");
      var dogsParty = new Party<Dog>(jake);
```

The `dogsParty` instance will only accept the `Dog` instance for all the arguments in which the class definition used the generic type parameter named `T`. The following code adds the previously created three instances of `Dog` to the dogs' party by calling the `AddMember` method:

```
dogsParty.AddMember(duke);
dogsParty.AddMember(lady);
dogsParty.AddMember(dakota);
```

The following code calls the `Dance` method to make all the dogs dance, removes a member who isn't the party leader, votes for a new leader, and finally calls the `Sing` method to make all the dogs sing:

```
dogsParty.Dance();
dogsParty.RemoveMember(duke);
dogsParty.VoteLeader();
dogsParty.Sing();
```

We can use `Frog` to create an instance of `Party<Frog>`. The following code creates four instances of the `Frog` class: `frog1`, `frog2`, `frog3`, and `frog4`. Then, the code creates a `Party<Frog>` instance named `frogsParty` and passes `frog1` as the argument. This way, we create the party of frogs in which `Frog #1` is the party leader:

```
var frog1 = new Frog("Frog #1");
var frog2 = new Frog("Frog #2");
var frog3 = new Frog("Frog #3");
var frog4 = new Frog("Frog #4");
var frogsParty = new Party<Frog>(frog1);
```

The `frogsParty` instance will only accept the `Frog` instance for all the arguments in which the class definition used the generic type parameter named `T`. The following code adds the previously created three instances of `Frog` to the frogs' party by calling the `AddMember` method:

```
frogsParty.AddMember(frog2);
frogsParty.AddMember(frog3);
frogsParty.AddMember(frog4);
```

The following code calls the `Dance` method to make all the frogs dance, removes a member who isn't the party leader, votes for a new leader, and finally calls the `Sing` method to make all the frogs sing:

```
frogsParty.Dance();
frogsParty.RemoveMember(frog3);
frogsParty.VoteLeader();
```

```
            frogsParty.Sing();
            Console.ReadLine();
        }
    }
```

The following lines show the console output after we run the preceding code snippets:

```
Jake welcomes Dakota: Wooooof
Jake dances /-\ \-\ /-/
Duke dances /-\ \-\ /-/
Lady dances /-\ \-\ /-/
Dakota dances /-\ \-\ /-/
Duke says goodbye to Jake: Woof Grr Woof
Jake says: Lady has been voted as our new party leader.
Lady dances /-\ \-\ /-/
Jake sings: Woof Woof Woof . Woof Woof . Woof .
Lady sings: Woof Woof Woof . Woof Woof . Woof .
Dakota sings: Woof Woof Woof . Woof Woof . Woof .
Frog #1 welcomes Frog #2: Croak
Frog #1 welcomes Frog #3: Croak
Frog #1 welcomes Frog #4: Croak
Frog #1 dances /|\ \|/ ^ ^
Frog #2 dances /|\ \|/ ^ ^
Frog #3 dances /|\ \|/ ^ ^
Frog #4 dances /|\ \|/ ^ ^
Frog #3 says goodbye to Frog #1: Ribbit Croooaaak Ribbit
Frog #1 says: Frog #2 has been voted as our new party leader.
Frog #2 dances /|\ \|/ ^ ^
Frog #1 sings: Ribbit Ribbit Ribbit . Ribbit Ribbit . Ribbit .
Frog #2 sings: Ribbit Ribbit Ribbit . Ribbit Ribbit . Ribbit .
Frog #4 sings: Ribbit Ribbit Ribbit . Ribbit Ribbit . Ribbit .
```

Declaring a class that works with two constrained generic types

Now, it is time to code another interface that will be used as a constraint later when we define another class that takes advantage of generics with two constrained generic types. The following lines show the code for the `IDeeJay` interface in C#. The `public` modifier, followed by the `interface` keyword and the `IDeeJay` interface name composes the interface declaration:

```
public interface IDeeJay
{
  string Name { get; set; }

  void PlayMusicToDance();
  void PlayMusicToSing();
}
```

The `IDeeJay` interface declares the `Name` string property and two methods: `PlayMusicToDance` and `PlayMusicToSing`. This interface includes only the method declaration because the classes that implement the `IDeeJay` interface will be responsible for providing the implementation of the getter and setter methods for the `Name` property and the other two methods.

Now, we will declare a class named `HorseDeeJay` that implements the previously defined `IDeeJay` interface. We can read the class declaration as, "the `HorseDeeJay` class implements the `IDeeJay` interface":

```
public class HorseDeeJay: IDeeJay
{
  protected string _name;

  public string Name
  {
    get { return this._name; }
    set { throw new InvalidOperationException("Name is a read-only
property."); }
  }

  public HorseDeeJay(string name)
  {
    this._name = name;
  }

  public void PlayMusicToDance()
```

```
    {
      Console.WriteLine(
      String.Format(
      "My name is {0}. Let's Dance.",
      this.Name));
    }

    public void PlayMusicToSing()
    {
      Console.WriteLine("Time to sing!");
    }
  }
```

The `HorseDeeJay` class declares a constructor that assigns the value of the required `name` argument to the `_name` protected field. The class declares the `Name` read-only property that encapsulates a related field. As happened with the `IAnimal` interface, the `HorseDeeJay` interface requires the `Name` property; therefore, it is necessary to create both setter and getter public methods. We cannot use auto-implemented properties with a private setter method because the setter method must be public. Thus, we defined the public setter method that throws an `InvalidOperationException` to avoid users of subclasses of this abstract class to change the value of the `Name` property.

The `PlayMusicToDance` method prints a message that displays the horse DJ name and invites all the party members to dance. The `PlayMusicToSing` method prints a message that invites all the party members to sing.

The following code declares the subclass of the previously created `Party<T>` class that takes advantage of generics to work with two constrained types. The class name is followed by a less than sign (<), `T` that identifies the generic type parameter, a comma (`,`), `K` that identifies the second generic type parameter, and a greater than sign (>). The first `where` keyword, followed by `T` that identifies the first type and a colon (`:`) indicates that the `T` generic type parameter has to be a type that implements the specified interface, that is, the `IAnimal` interface. The second `where` keyword, followed by `K` that identifies the second type and a colon (`:`) indicates that the `K` generic type parameter has to be a type that implements the specified interface, that is, the `IDeeJay` interface. This way, the `Party<T>` class specifies constraints for the `T` and `K` generic type parameters. Don't forget that we are talking about a subclass of `Party<T>`:

```
public class PartyWithDeeJay<T, K>: Party<T> where T: IAnimal
where K: IDeeJay
```

The following line starts the class body and declares the public `DeeJay` auto-implemented property of the type specified by `K`:

```
{
  public K DeeJay { get; private set; }
```

The following code declares a constructor that receives two arguments: `leader` and `deejay`, whose types are `T` and `K`. These arguments specify the first party leader, the first member of the party, and the DJ that will make all the party members dance and sing. Note that the constructor calls the base constructor, that is, the `Party<T>` constructor with `leader` as the argument:

```
  public PartyWithDeeJay(T leader, K deeJay): base(leader)
  {
    this.DeeJay = deeJay;
  }
```

The following code declares the `Dance` method that overrides the method with the same declaration included in the superclass. The code calls the `DeeJay.PlayMusicToDance` method and then calls the `base.Dance` method, that is, the `Dance` method defined in the `Party<T>` superclass:

```
  public override void Dance()
  {
    this.DeeJay.PlayMusicToDance();
    base.Dance();
  }
```

Finally, the following code declares the `Sing` method that overrides the method with the same declaration included in the superclass. The code calls the `DeeJay.PlayMusicToSing` method and then calls the `base.Sing` method, that is, the `Sing` method defined in the `Party<T>` superclass:

```
  public override void Sing()
  {
    this.DeeJay.PlayMusicToSing();
    base.Sing();
  }
}
```

Using a generic class with two generic type parameters

We can create instances of the `PartyWithDeeJay<T, K>` class by replacing both the `T` and `K` generic type parameters with any type names that conform to the constraints specified in the declaration of the `PartyWithDeeJay<T, K>` class. We have two concrete classes that implement the `IAnimal` interface: `Dog` and `Frog` and one class that implements the `IDeeJay` interface: `HorseDeeJay`. Thus, we can use `Dog` and `HorseDeeJay` to create an instance of `PartyWithDeeJay<Dog, HorseDeeJay>`.

The following is the code that we can add to our previously created console application to create the `HorseDeeJay` instance named `silver`. Then, the code creates the `PartyWithDeeJay<Dog, HorseDeeJay>` instance named `silverParty` and passes `jake` and `silver` as arguments. This way, we can create a party with a dog leader and a horse DJ, where Jake is the party leader and Silver is the DJ:

```
var silver = new HorseDeeJay("Silver");
var silverParty = new PartyWithDeeJay<Dog, HorseDeeJay>(jake, silver);
```

The `silverParty` instance will only accept the `Dog` instance for all the arguments in which the class definition uses the generic type parameter named `T`. The following code adds the previously created three instances of `Dog` to the party by calling the `AddMember` method:

```
silverParty.AddMember(duke);
silverParty.AddMember(lady);
silverParty.AddMember(dakota);
```

The following code calls the `Dance` method to make the DJ invite all the dogs to dance and then make them dance. Then, the code removes a member who isn't the party leader, votes for a new leader, and finally calls the `Sing` method to make the DJ invite all the dogs to sing and then make them sing:

```
silverParty.Dance();
silverParty.RemoveMember(duke);
silverParty.VoteLeader();
silverParty.Sing();
```

The following lines display the console output after we run the added code:

```
My name is Silver. Let's Dance.
Jake dances /-\ \-\ /-/
Duke dances /-\ \-\ /-/
Lady dances /-\ \-\ /-/
Dakota dances /-\ \-\ /-/
```

```
Duke says goodbye to Jake: Woof Grr Woof
Jake says: Lady has been voted as our new party leader.
Lady dances /-\ \-\ /-/
Time to sing!
Jake sings: Woof Woof Woof . Woof Woof . Woof .
Lady sings: Woof Woof Woof . Woof Woof . Woof .
Dakota sings: Woof Woof Woof . Woof Woof . Woof .
```

Working with duck typing in JavaScript

We will create an `Animal` constructor function and customize its prototype to generalize all the requirements for animals. We will create two constructor functions: `Dog` and `Frog` that will use `Animal` as its prototype. Then, we will create a `Party` constructor function that will be able to work with instances of any object that includes `Animal` in its prototype chain through duck typing. We will work with the party of dogs and the party of frogs.

Then, we will create a `HorseDeeJay` constructor function and generate a new version of the `Party` constructor function that will work with any object that includes `Animal` in its prototype and any object that provides the properties and methods declared in the `HorseDeeJay` prototype through duck typing. We will work with the party of dogs with a DJ.

Declaring a constructor function that defines the generic behavior

Now, we will declare the `Animal` constructor function. Then, we will add properties and methods to its prototype:

```
Animal = function() { };
Animal.prototype.name = "";
Animal.prototype.danceCharacters = "";
Animal.prototype.spelledSound1 = "";
Animal.prototype.spelledSound2 = "";
Animal.prototype.spelledSound3 = "";

Animal.prototype.dance = function() {
  console.log(this.name + " dances " + this.danceCharacters);
}

Animal.prototype.say = function(message) {
  console.log(this.name + " says: " + message);
}

Animal.prototype.sayGoodbye = function(destination) {
  console.log(this.name + " says goodbye to "
  + destination.name + ": "
  + this.spelledSound1 + " "
  + this.spelledSound2 + " "
  + this.spelledSound3 + " ");
}

Animal.prototype.sayWelcome = function(destination) {
  console.log(this.name + " welcomes "
  + destination.name + ": "
  + this.spelledSound2);
}

Animal.prototype.sing = function() {
  var spelledSingSound = this.spelledSound1 + " ";
  var message = this.name + " sings: " + Array(4).
join(spelledSingSound) + ". " + Array(3).join(spelledSingSound) + ". "
+ spelledSingSound + ". ";

  console.log(message);
}
```

The preceding code declared the following five properties for the prototype (all of them initialized with an empty string). The constructor functions that will use `Animal` as its prototype will override these properties with the appropriate strings according to the animal. These constructor functions will receive the value to be set to `name` as the argument:

- `name`
- `danceCharacters`
- `spelledSound1`
- `spelledSound2`
- `spelledSound3`

The `dance` method uses the value retrieved from the `danceCharacters` property to print a message. This message indicates that the animal is dancing. The `say` method prints the message received as an argument.

Both the sayWelcome and sayGoodbye methods receive the destination argument that they use to print the name of the destination of the message. Thus, whenever we call this method, the destination argument must be the object that has the name property to be considered an animal. Any instance of any constructor function that has Animal as its prototype qualifies as the destination argument for both methods.

The sayWelcome method uses a combination of strings retrieved from the spelledSound1 and spelledSound3 properties to say welcome to another animal. The sayGoodbye method uses the string retrieved from the spelledSound2 property to say goodbye to another animal.

Working with the prototype chain and duck typing

Now, we will create a constructor function that will use Animal as its prototype. The Dog constructor function overrides the string properties defined in the Animal prototype to provide all the appropriate values for a dog. The constructor function receives name as the argument and assigns its value to the property with the same name:

```
Dog = function(name) {
    this.name = name;
};
Dog.prototype = new Animal();
Dog.prototype.constructor = Dog;
Dog.prototype.danceCharacters = "/-\\ \\-\\ /-/";
Dog.prototype.spelledSound1 = "Woof";
Dog.prototype.spelledSound2 = "Wooooof";
Dog.prototype.spelledSound3 = "Grr";
```

With just a few additional lines of code, we will create another constructor function that uses Animal as its prototype. The Frog constructor function also overrides the string class attributes defined in the Animal constructor function to provide all the appropriate values for a frog:

```
Frog = function(name) {
  this.name = name;
};
Frog.prototype = new Animal();
Frog.prototype.constructor = Frog;
Frog.prototype.danceCharacters = "/|\\ \\|/ ^ ^ ";
Frog.prototype.spelledSound1 = "Ribbit";
Frog.prototype.spelledSound2 = "Croak";
Frog.prototype.spelledSound3 = "Crooooaaak";
```

Declaring methods that work with duck typing

The following code declares the `Party` constructor function that takes advantage of duck typing to work with instances of any class that provides the `name` property and implements the `dance`, `say`, `sayGoodbye`, `sayWelcome`, and `sing` methods. This constructor function receives a `leader` argument that the code assigns to the `leader` property. In addition, the code creates a `leader` array as one of its members and saves it in the `members` property. This way, the `leader` argument specifies the first party leader and also the first member of the party, that is, the first element added to the `members` array:

```
Party = function(leader) {
  this.leader = leader;
  this.members = [leader];
}
```

The following code declares the `addMember` method that receives the `member` argument. The code adds the member received as an argument to the `members` array and calls the `leader.sayWelcome` method with `member` as the argument to make the party leader welcome the new member:

```
Party.prototype.addMember = function(member) {
  this.members.push(member);
  this.leader.sayWelcome(member);
}
```

The following code declares the `removeMember` method that receives the `member` argument. The code checks whether the member to be removed is the party leader. The method throws an exception if the member is the party leader. If the member isn't the party leader, the code removes the member from the `members` array and calls the `sayGoodbye` method for the removed member. This way, the member who leaves the party says goodbye to the party leader:

```
Party.prototype.removeMember = function(member) {
  if (member == this.leader) {
    throw "You cannot remove the leader from the party";
  }
  var index = this.members.indexOf(member);
  if (index > -1) {
    this.members.splice(index, 1);
    member.sayGoodbye(this.leader);
    return true;
  }
  else {
    return false;
  }
}
```

The following code declares the `dance` method that calls the method with the same name for each member of the `members` array:

```
Party.prototype.dance = function() {
  this.members.forEach(function (member) { member.dance(); });
}
```

The following code declares the `sing` method that calls the method with the same name for each member of the `members` array:

```
Party.prototype.sing = function() {
  this.members.forEach(function (member) { member.sing(); });
}
```

Finally, the following code declares the `voteLeader` method. The code makes sure that there are at least two members in the `members` array when we call this method; if we have just one member, the method throws an exception. If we have at least two members, the code generates a new pseudo-random leader (who is different from the existing leader) for the party. The code calls the `say` method for the actual leader to make it explain to the other party members that another leader has been voted for. Finally, the code calls the `dance` method for the new leader and sets the new value for the `leader` property:

```
Party.prototype.voteLeader = function() {
  if (this.members.length == 1) {
    throw "You need at least two members to vote a new Leader.";
  }
  var newLeader = this.leader;
  while (newLeader == this.leader) {
    var randomLeader = Math.floor(Math.random() * (this.members.length
- 1)) + 1;
    newLeader = this.members[randomLeader];
  }
  this.leader.say(newLeader.name + " has been voted as our new party
leader.");
  newLeader.dance();
  this.leader = newLeader;
}
```

Using generic methods for multiple objects

We have two constructor functions: `Dog` and `Frog` that use the `Animal` constructor function as their prototype. Both these constructor functions define all the properties and methods required that allows you to work with their instances as arguments of the methods of the previously coded `Party` prototype methods. We can start working with `Dog` objects to create the party of dogs.

The following code creates four `Dog` objects: `jake`, `duke`, `lady` and `dakota`:

```
var jake = new Dog("Jake");
var duke = new Dog("Duke");
var lady = new Dog("Lady");
var dakota = new Dog("Dakota");
```

The following line creates a `Party` object named `dogsParty` and passes `jake` as the argument. This way, we create the party of dogs in which Jake is the party leader:

```
var dogsParty = new Party(jake);
```

The following code adds the previously created three `Dog` objects to the dogs' party by calling the `addMember` method:

```
dogsParty.addMember(duke);
dogsParty.addMember(lady);
dogsParty.addMember(dakota);
```

The following code calls the `dance` method to make all the dogs dance, removes a member who isn't the party leader, votes for a new leader, and finally calls the `sing` method to make all the dogs sing:

```
dogsParty.dance();
dogsParty.removeMember(duke);
dogsParty.voteLeader();
dogsParty.sing();
```

The following lines display the output generated on the JavaScript console after running the preceding code snippets:

```
Jake welcomes Duke: Wooooof
Jake welcomes Lady: Wooooof
Jake welcomes Dakota: Wooooof
Jake dances /-\ \-\ /-/
Duke dances /-\ \-\ /-/
Lady dances /-\ \-\ /-/
Dakota dances /-\ \-\ /-/
```

```
Duke says goodbye to Jake: Woof Wooooof Grr
Jake says: Dakota has been voted as our new party leader.
Dakota dances /-\ \-\ /-/
Jake sings: Woof Woof Woof . Woof Woof . Woof .
Lady sings: Woof Woof Woof . Woof Woof . Woof .
Dakota sings: Woof Woof Woof . Woof Woof . Woof .
```

Now, what about the party of frogs? The following code creates four `Frog` objects: `frog1`, `frog2`, `frog3`, and `frog4`:

```
var frog1 = new Frog("Frog #1");
var frog2 = new Frog("Frog #2");
var frog3 = new Frog("Frog #3");
var frog4 = new Frog("Frog #4");
```

The following line creates a `Party` object named `frogsParty` and passes `frog1` as the argument. This way, we create the party of dogs in which `Frog #1` is the party leader:

```
var frogsParty = new Party(frog1);
```

The following code adds the previously created three `Frog` objects to the frogs' party by calling the `addMember` method:

```
frogsParty.addMember(frog2);
frogsParty.addMember(frog3);
frogsParty.addMember(frog4);
```

The following code calls the `dance` method to make all the frogs dance, removes a member who isn't the party leader, votes for a new leader, and finally calls the `sing` method to make all the frogs sing:

```
frogsParty.dance();
frogsParty.removeMember(frog3);
frogsParty.voteLeader();
frogsParty.sing();
```

The following lines display the output generated on the JavaScript console after running the previous code:

```
Frog #1 welcomes Frog #2: Croak
Frog #1 welcomes Frog #3: Croak
Frog #1 welcomes Frog #4: Croak
Frog #1 dances /|\ \|/ ^ ^
Frog #2 dances /|\ \|/ ^ ^
Frog #3 dances /|\ \|/ ^ ^
Frog #4 dances /|\ \|/ ^ ^
Frog #3 says goodbye to Frog #1: Ribbit Croak Croooaaak
Frog #1 says: Frog #2 has been voted as our new party leader.
Frog #2 dances /|\ \|/ ^ ^
Frog #1 sings: Ribbit Ribbit Ribbit . Ribbit Ribbit . Ribbit .
Frog #2 sings: Ribbit Ribbit Ribbit . Ribbit Ribbit . Ribbit .
Frog #4 sings: Ribbit Ribbit Ribbit . Ribbit Ribbit . Ribbit .
```

Working with duck typing in mind

Now, we will create a new constructor function that declares properties and methods that we will call from an object that includes `Party` in its prototype chain. As long as we use objects that provide all the required properties and methods, we can use all the objects created by using any constructor function with the new object. The following lines show the code for the `HorseDeeJay` constructor function:

```
HorseDeeJay = function(name) {
  this.name = name;
};
HorseDeeJay.prototype.playMusicToDance = function() {
  console.log("My name is " + this.name + ". Let's Dance.");
}
HorseDeeJay.prototype.playMusicToSing = function() {
  console.log("Time to sing!");
}
```

The `HorseDeeJay` constructor function class assigns the value of the `name` argument to the `name` property. The `playMusicToDance` method prints a message that displays the horse DJ name and invites all the party members to dance. The `playMusicToSing` method prints a message that invites all the party members to sing.

Now, we will make a few changes to the previously created `Party` constructor function and its prototype to allow it to work with a `HorseDeeJay` object. The following code shows the new version of the constructor function that receives two arguments: `leader` and `deejay`. These arguments specify the first party leader, the first member of the party, and the DJ that will make all the party members dance and sing:

```
Party = function(leader, deeJay) {
  this.leader = leader;
  this.deeJay = deeJay;
  this.members = [leader];
}
```

The following code declares the new version of the `dance` method. The code calls the `deeJay.playMusicToDance` method and then calls the `dance` method for each member of the party:

```
Party.prototype.dance = function() {
  this.deeJay.playMusicToDance();
  this.members.forEach(function (member) { member.dance(); });
}
```

Finally, the following code declares the new version of the `sing` method. The code calls the `deeJay.playMusicToSing` method and then calls the `sing` method for each member of the party:

```
Party.prototype.sing = function() {
  this.deeJay.playMusicToSing();
  this.members.forEach(function (member) { member.sing(); });
}
```

The other methods defined for the `Party` prototype are the same that we defined in the previous version. The following code shows the lines that we can run on the JavaScript console to create a `HorseDeeJay` object named `silver`. Then, the code creates a `PartyWithDeeJay` object named `silverParty` and passes `jake` and `silver` as arguments. This way, we create a party with a dog leader and a horse DJ in which Jake is the party leader and Silver is the DJ:

```
var silver = new HorseDeeJay("Silver");
var silverParty = new Party(jake, silver);
```

The following code adds the previously created three `Dog` objects to the party by calling the `addMember` method:

```
silverParty.addMember(duke);
silverParty.addMember(lady);
silverParty.addMember(dakota);
```

The following code calls the `dance` method to make the DJ invite all the dogs to dance and then make them dance. Then, the code removes a member who isn't the party leader, votes for a new leader, and finally calls the `sing` method to make the DJ invite all the dogs to sing and then make them sing:

```
silverParty.dance();
silverParty.removeMember(duke);
silverParty.voteLeader();
silverParty.sing();
```

The following lines display the JavaScript console output after we run the added code:

```
My name is Silver. Let's Dance.
Jake dances /-\ \-\ /-/
Duke dances /-\ \-\ /-/
Lady dances /-\ \-\ /-/
Dakota dances /-\ \-\ /-/
Duke says goodbye to Jake: Woof Wooooof Grr
Jake says: Dakota has been voted as our new party leader.
Dakota dances /-\ \-\ /-/
Time to sing!
Jake sings: Woof Woof Woof . Woof Woof . Woof .
Lady sings: Woof Woof Woof . Woof Woof . Woof .
Dakota sings: Woof Woof Woof . Woof Woof . Woof .
```

Summary

In this chapter, you learned how to maximize code reuse by writing code capable of working with objects of different types. We took advantage of duck typing in Python and JavaScript. We worked with interfaces and generics in C#. We created classes capable of working with one and two constrained generic types.

Now that you learned how to work with duck typing and generics, we are ready to organize complex object-oriented code in Python, JavaScript, and C#, which is the topic of the next chapter.

7
Organization of Object-Oriented Code

In this chapter, we will write code for a complex application that requires dozens of classes, interfaces, and constructor functions according to the programing language that we use. We will take advantage of the different available features to organize a large number of pieces of code in each of the three covered programming languages: Python, JavaScript, and C#. We will:

- Understand the importance of organizing object-oriented code
- Think about the best ways to organize object-oriented code
- Work with source files organized in folders and module hierarchies in Python
- Work with folders, namespaces, and namespace hierarchies in C#
- Combine objects, nested objects, and constructor functions in JavaScript

Thinking about the best ways to organize code

When you have just a few classes or constructor functions and their prototypes, hundreds of lines of object-oriented code are easy to organize and maintain. However, as the number of object-oriented blueprints start to increase, it is necessary to follow some rules to organize the code and make it easy to maintain.

A very well written object-oriented code can generate a maintenance headache if it isn't organized in an effective way. We don't have to forget that a well written object-oriented code promotes code reuse.

As you learned in the previous six chapters, each programming language provides different elements and resources to generate object-oriented code. In addition, each programming language provides its own mechanisms that allow you to organize and group different object-oriented elements. Thus, it is necessary to define rules for each of the three programming languages: Python, C# and, JavaScript.

Imagine that we have to create and furnish house floor plans with a drawing software that allows you to load objects from files. We have a huge amount of objects to compose our floor plan, such as entry doors, interior doors, square rooms, interior walls, windows, spiral stairs, straight stairs, and kitchen islands. If we use a single folder in our file system to save all the object files, it will take us a huge amount of time to select the desired object each time we have to add an object to our floor plan.

We can organize our objects in the following five folders:

- `Build`
- `Furnish`
- `Decorate`
- `Landscape`
- `Outdoor`

Now, whenever we need bathroom furniture, we will explore the `Furnish` folder. Whenever we need outdoor structures, we will explore the `Outdoor` folder. However, there are still too many objects in each of these folders. For example, the `Build` folder includes the following types of objects:

- `Rooms`
- `Walls`
- `Areas`
- `Doors`
- `Windows`
- `Stairs`
- `Fireplaces`

We can create subfolders within each main category folder to provide a better organization of our object files. The `Build` category will have one subfolder for each of the types of objects indicated in the previous list.

The `Furnish` category will have the following subfolders:

- `Living room`
- `Dining room`
- `Kitchen`
- `Bathroom`
- `Bedroom`
- `Office`
- `Laundry and utility`
- `Other rooms`

The `Decorate` category will have the following subfolders:

- `Paint and walls`
- `Flooring`
- `Countertops`
- `Art and decor`
- `Electronics`
- `Lighting and fans`

The `Landscape` category will have the following subfolders:

- `Areas definition`
- `Materials`
- `Trees and plants`

Finally, the `Outdoor` category will have the following subfolders:

- `Living`
- `Accessories`
- `Structures`

This way, the `Build/Rooms` subfolder will include the following four objects:

- `Square room`
- `L-shaped room`
- `Small room`
- `Closet`

However, the `Furnish/Bedroom` subfolder includes too many objects that we can organize in seven types. So, we will create the following six subfolders:

- `Beds`
- `Kids' beds`
- `Night tables`
- `Dressers`
- `Mirrors`
- `Nursery`

Whenever we need bedroom mirrors, we will go to the `Furnish/Bedroom/Mirrors` subfolder. Whenever we need beds, we will go to the `Furnish/Bedroom/Beds` subfolder. Our objects are organized in a hierarchical directory tree.

Now, let's go back to object-oriented code. Instead of objects, we will have to organize classes, interfaces, constructor functions, and prototypes according to the programming language used. For example, if we have a class that defines the blueprint for a square room, we can organize it in such a way that we can find it in a `build.rooms` container. This way, we will find all the classes related to `Build/Rooms` in the `build.rooms` container. If we need to add another class related to `Build/Rooms`, we would add it in the `build.rooms` container.

Organizing object-oriented code in Python

Python makes it easy to logically organize object-oriented code with modules. We will work with a hierarchy of folders to organize the code of an application that allows you to create and furnish house floor plans. Then, we will use code from different folders and the source files of Python.

Working with source files organized in folders

We will create the following six folders to organize the code in our house floor plan layout application. Then, we will add subfolders and the source files of Python to each of the previously created folders:

- `Build`
- `Decorate`
- `Furnish`
- `General`

- `Landscape`
- `Outdoor`

We will include all the base classes in the `general` folder. The following lines show the code for the `general/floor_plan_element.py` Python source file that declares a `FloorPlanElement` base class. We will use this class as the superclass for all the classes that specialize the floor plan element:

```
class FloorPlanElement:
    category = "Undefined"
    description = "Undefined"

    def __init__(self, x, y, width, height, parent):
        self.x = x
        self.y = y
        self.width = width
        self.height = height
        self.parent = parent

    def move_to(self, x, y):
        self.x = x
        self.y = y

    def print_category(self):
        print(type(self).category)

    def print_description(self):
        print(type(self).description)

    def draw(self):
        self.print_category()
        self.print_description()
        print("X: " + str(self.x) +
              ", Y: " + str(self.y) +
              ". Width: " + str(self.width) +
              ", Height: " + str(self.height) + ".")
```

The `FloorPlanElement` class declares two class attributes that the subclasses will override: `category` and `description`. The class declares an `__init__` method that receives five arguments: `x`, `y`, `width`, `height`, and `parent`. The `__init__` method initializes attributes with the same name using all the values received as arguments. This way, each `FloorPlanElement` instance will have a 2D location specified by `x` and `y`, `width` and `height`, and a `parent` element.

In addition, the `FloorPlanElement` class declares the following instance methods:

- `move_to`: This method moves the floor plan element to the new location specified by the `x` and `y` arguments
- `print_category`: This method prints the value of the `category` class attribute
- `print_description`: This method prints the value of the `description` class attribute
- `draw`: This method prints the category, description, 2D location, and width and height of the floor plan element

The `FloorPlanElement` class is located in the `general/floor_plan_element.py` file. All the classes that specialize floor plan elements will inherit from the `FloorPlanElement` class. These classes will be located in other Python source files that will have to let Python know that they want to use the `FloorPlanElement` class that is located in another module, that is, in another Python source file.

Importing modules

We will create the following Python source files in the previously created `Build` folder. We won't add code to all the files in order to keep our example simple. However, we will imagine that we have a more complex project:

- `areas.py`
- `doors.py`
- `fireplaces.py`
- `rooms.py`
- `stairs.py`
- `walls.py`
- `windows.py`

Here is the code for the `build/rooms.py` Python source file that declares five classes: `Room`, `SquareRoom`, `LShapedRoom`, `SmallRoom`, and `Closet`:

```
from general.floor_plan_element import FloorPlanElement


class Room(FloorPlanElement):
    category = "Room"


class SquareRoom(Room):
    description = "Square room"

    def __init__(self, x, y, width, parent):
        super().__init__(x, y, width, width, parent)


class LShapedRoom(Room):
    description = "L-Shaped room"


class SmallRoom(Room):
    description = "Small room"


class Closet(Room):
    description = "Closet"
```

The first line in the preceding code uses the `from` statement to import a specific class from a module into the current namespace. To be specific, the code imports the `FloorPlanElement` class from the `general.floor_plan_element` module, that is, from the `general/floor_plan_element.py` file, as shown in the following line:

```
from general.floor_plan_element import FloorPlanElement
```

This way, the next few lines declare the `Room` class as a subclass of the `FloorPlanElement` class as if the class were defined in the same Python source file. This class overrides the value of the category class attribute with the `"Room"` value. The `SquareRoom` class represents a square room; therefore, it isn't necessary to specify the width and height to create an instance of this type of room. The `__init__` method uses the `width` value to specify the values for both width and height in the call to the `__init__` method defined in the superclass, that is, the `FloorPlanElement` class. Each subclass of the `Room` class overrides the `description` class attribute with an appropriate value.

In this case, the `from` statement combined with the `import` statement imported just one class definition into the current namespace. We can use the following import statement to import all the items from the `general.floor_plan_element` module, that is, from the `general/floor_plan_element.py` file. However, we must be careful when we specify `*` after the import statement because we should import only the elements that we need from the other module:

```
from general.floor_plan_element import *
```

Another option is to execute an import statement in order to import the required module. However, we would need to make changes to the Room class declaration and add the modules path as a prefix to FloorPlanElement, that is, we have to replace FloorPlanElement with general.floor_plan_element.FloorPlanElement, as shown in the following code:

```
import general.floor_plan_element


class Room(general.floor_plan_element.FloorPlanElement):
    category = "Room"
```

Here is the code for the build/doors.py Python source file that declares two classes: Door and EntryDoor:

```
from general.floor_plan_element import FloorPlanElement


class Door(FloorPlanElement):
    category = "Door"


class EntryDoor(Door):
    description = "Entry Door"
```

The first line in the preceding code uses the from statement to import the FloorPlanElement class from the general.floor_plan_element module, that is, from the general/floor_plan_element.py file. The next few lines declare the Door class as a subclass of the FloorPlanElement class as if the class were defined in the same Python source file. The FloorPlanElement class overrides the value of the category class attribute with the "Door" value. The EntryDoor class represents an entry door and just overrides the description class attribute with an appropriate value.

We will create the following Python source files in the previously created Decorate folder:

- art_and_decor.py
- countertops.py
- electronics.py
- flooring.py
- lighting_and_fans.py
- paint_and_walls.py

We will create the following Python source files in the previously created `Furnish` folder:

- `bathroom.py`
- `dining_room.py`
- `kitchen.py`
- `laundry_and_utility.py`
- `living_room.py`
- `office.py`
- `other_rooms.py`

We will create a `Bedroom` subfolder within the `Furnish` folder. Then, we will create the following Python source files within the `Furnish`/`Bedroom` subfolder:

- `beds.py`
- `dressers.py`
- `kids_beds.py`
- `mirrors.py`
- `night_tables.py`
- `nursery.py`

The following lines show the code for the `Furnish/Bedroom/beds.py` Python source file that declares two classes: `Bed` and `FabricBed`:

```
from general.floor_plan_element import FloorPlanElement


class Bed(FloorPlanElement):
    category = "Bed"
    description = "Generic bed"


class FabricBed(Bed):
    description = "Fabric bed"
```

The first line in the preceding code uses the `from` statement to import the `FloorPlanElement` class from the `general.floor_plan_element` module, that is, from the `general/floor_plan_element.py` file. The next few lines declare the `Bed` class as a subclass of the `FloorPlanElement` class as if the class were defined in the same Python source file. The `FloorPlanElement` class overrides the value of the `category` class attribute with the `"Bed"` value and `description` with `"Generic Bed"`. The `FabricBed` class represents a fabric bed and just overrides the `description` class attribute with an appropriate value.

We will create the following Python source files in the previously created `Landscape` folder:

- `areas_definition.py`
- `materials.py`
- `trees_and_plants.py`

Finally, we will create the following Python source files in the previously created `Outdoor` folder:

- `accessories.py`
- `living.py`
- `structures.py`

Working with module hierarchies

Now, we will create the `__main__.py` Python source file in the project's root folder, that is, the same folder that includes the following subfolders: `Build`, `Decorate`, `Furnish`, `General`, `Landscape`, and `Outdoor`. The following is the code that imports many of the previously defined modules and works with instances of all the imported classes:

```
from build.rooms import *
from build.doors import *
from furnish.bedroom.beds import *

if __name__ == "__main__":
    room1 = SquareRoom(0, 0, 200, None)
    door1 = EntryDoor(100, 1, 50, 5, room1)
    bedroom1 = SquareRoom(100, 200, 180, None)
    bed1 = FabricBed(130, 230, 120, 110, bedroom1)
    room1.draw()
    door1.draw()
    bedroom1.draw()
    bed1.draw()
```

The first line in the preceding code uses the `from` statement to import all the classes from the following modules into the current namespace:

- `build.rooms`: `build/rooms.py`
- `build.doors`: `build/doors.py`
- `furnish.bedroom.beds`: `furnish/bedrooms/beds.py`

This way, we can access the `SquareRoom`, `EntryDoor` and `FabricBed` classes as if they were defined in the `__main__.py` Python source file. The following is the output generated with the preceding code:

```
Room
Square room
X: 0, Y: 0. Width: 200, Height: 200.
Door
Entry Door
X: 100, Y: 1. Width: 50, Height: 5.
Room
Square room
X: 100, Y: 200. Width: 180, Height: 180.
Bed
Fabric bed
X: 130, Y: 230. Width: 120, Height: 110.
```

In this case, we just need these three classes; therefore, we can use the following `from` statements to import just the classes we need:

```
from build.rooms import SquareRoom
from build.doors import EntryDoor
from furnish.bedroom.beds import FabricBed
```

Another option is to use the `import` keyword and add the necessary modules hierarchy separated by dots (`.`) to each class defined in modules. The following lines show the version of the code that uses the `import` keyword and adds the necessary prefixes to each class:

```
import build.doors
import build.rooms
import furnish.bedroom.beds

if __name__ == "__main__":
    room1 = build.SquareRoom(0, 0, 200, None)
    door1 = build.EntryDoor(100, 1, 50, 5, room1)
```

```
        bedroom1 = build.SquareRoom(100, 200, 180, None)
        bed1 = furnish.bedroom.beds.FabricBed(130, 230, 120, 110,
    bedroom1)
        room1.draw()
        door1.draw()
        bedroom1.draw()
        bed1.draw()
```

Now, let's imagine that we have to code an application that has to draw the floor plans for 30 extremely complex houses that will be displayed in a 4K display. We will have to work with most of the elements defined in all the different modules. We would like to have all the classes defined in the Python source files included in the `Build` folder with just one line. It is possible to do so by adding the `__init__.py` Python source file to the `Build` folder and including the following code to import all the classes defined in each of the Python source files included in this folder:

```
from .areas import *
from .doors import *
from .fireplaces import *
from .rooms import SquareRoom, LShapedRoom, SmallRoom, Closet
from .stairs import *
from .walls import *
from .windows import *
```

This way, if we use the `import` keyword with the `build` module, we will be able to access all the classes defined in each of the Python source files included in the `build` module by adding the `build.` prefix to each class name. In fact, we won't be able to access all the classes because we excluded the `Room` class in the `from .rooms import` statement and specified only four classes to import from this module. After we add the `__init__.py` Python source file to the `Build` folder, we can change the `__main__.py` Python source code with the following code:

```
import build
from furnish.bedroom.beds import FabricBed

if __name__ == "__main__":
    room1 = build.SquareRoom(0, 0, 200, None)
    door1 = build.EntryDoor(100, 1, 50, 5, room1)
    bedroom1 = build.SquareRoom(100, 200, 180, None)
    bed1 = FabricBed(130, 230, 120, 110, bedroom1)
    room1.draw()
    door1.draw()
    bedroom1.draw()
    bed1.draw()
```

If we don't want to use prefixes and just want to import all the classes defined in the build module, we can use the following `from … import` statement:

```
from build import *
```

We can add the appropriate code in the **`__init__.py`** Python source file for each folder and then use the appropriate **`import`** or **`from … import`** statements based on our needs.

Organizing object-oriented code in C#

C# allows you to use namespaces to declare a scope that contains a set of related elements. Thus, we can use namespaces to organize interfaces and classes. We will work with nested namespaces to organize the code of an application. This allows you to create and furnish house floor plans. Then, we will use interfaces and classes from different namespaces in diverse pieces of code.

Working with folders

We will create a Windows console application named `Chapter7`. Visual Studio will automatically add a `Program.cs` C# source file in the solution's root folder. We will create the following six folders to organize the code in our house floor plan layout application. Then, we will add subfolders and C# source files in each of the previously created folders. Visual Studio will use these folders and subfolders to automatically generate the namespaces for each new C# source file:

- `Build`
- `Decorate`
- `Furnish`
- `General`
- `Landscape`
- `Outdoor`

We will use the C# source file for each interface or class. We will add the `IFloorPlanElement` interface to a file named `IFloorPlanElement.cs` in the `General` folder. The following is the code for the `General\IFloorPlanElement.cs` C# source file that declares the `IFloorPlanElement` interface:

```
namespace Chapter7.General
{
  public interface IFloorPlanElement
  {
    string Category { get; set; }
    string Description { get; set; }
    double X { get; set; }
    double Y { get; set; }
    double Width { get; set; }
    double Height { get; set; }
    IFloorPlanElement Parent { get; set; }

    void MoveTo(double x, double y);
    void PrintCategory();
    void PrintDescription();
    void Draw();
  }
}
```

If we use Visual Studio to create a new interface in the `General` folder, the IDE will automatically include a line with the `namespace` keyword, followed by the `Chapter7.General` name. The declaration of the interface will be enclosed in brackets after the line that declares the namespace. The IDE uses the folder name in which we will add the interface to automatically generate a default namespace name. In this case, the generated name is the initial namespace, that is, the `Chapter7` solution name, followed by a dot (`.`) and the folder that contains the new C# source file, that is, `General`.

The `IFloorPlanElement` interface declares the following required members:

- Two string properties: `Category` and `Description`
- Four double properties: `X`, `Y`, `Width`, and `Height`
- The `IFloorPlanElement` property: `Parent`
- The `MoveTo` method that receives two arguments: `x` and `y`
- The `PrintCategory` method
- The `PrintDescription` method
- The `Draw` method

Now, we will add the `FloorPlanElement` abstract class that implements the previously created `IFloorPlanElement` interface to a file named `FloorPlanElement.cs` in the `General` folder. The following is the code for the `General\FloorPlanElement.cs` C# source file that declares the `FloorPlanElement` class:

```
namespace Chapter7.General
{
  using System;

  public abstract class FloorPlanElement: IFloorPlanElement
  {
    public virtual string Category
    {
      get { return "Undefined"; }
      set { throw new InvalidOperationException(); }
    }

    public virtual string Description
    {
      get { return "Undefined"; }
      set { throw new InvalidOperationException(); }
    }

    public double X { get; set; }
    public double Y { get; set; }
    public double Width { get; set; }
    public double Height { get; set; }

    private IFloorPlanElement _parent;
    public IFloorPlanElement Parent
    {
      get { return _parent; }
      set { throw new InvalidOperationException(); }
    }

    public FloorPlanElement(double x, double y, double width, double 
height, IFloorPlanElement parent)
    {
      this.X = x;
      this.Y = y;
      this.Width = width;
      this.Height = height;
      this._parent = parent;
```

```
        }

        public void MoveTo(double x, double y)
        {
          this.X = x;
          this.Y = y;
        }

        public void PrintCategory()
        {
          Console.WriteLine(this.Category);
        }

        public void PrintDescription()
        {
          Console.WriteLine(this.Description);
        }

        public void Draw()
        {
          this.PrintCategory();
          this.PrintDescription();
          Console.WriteLine(
          "X: {0}, Y: {1}. Width: {2}, Height: {3}.",
          this.X,
          this.Y,
          this.Width,
          this.Height);
        }
      }
    }
```

Note that the abstract class is declared in the same namespace that we used for the `IFloorPlanElement` interface; therefore, both the class and the interface are in the same scope. We can reference the interface name in the class declaration without any namespace prefix because both the class and the interface are declared in the same namespace.

The abstract class declares a constructor that receives five arguments: `x`, `y`, `width`, `height`, and `parent`. The preceding code initializes all the properties with the values received as arguments. This way, each `FloorPlanElement` instance will have a 2D location specified by the `X` and `Y` properties, a `Width` and a `Height` value, and a `Parent` element that will be of any class that implements the `IFloorPlanElement` interface.

The `FloorPlanElement` abstract class declares two `virtual` read-only properties that the subclasses will override: `Category` and `Description`. In addition, the abstract class declares all the other properties required by the `IFloorPlanElement` interface: `X`, `Y`, `Width`, `Height`, and `Parent`. Note that `Parent` is a read-only property that encapsulates the private `_parent` field.

The abstract class declares the following instance methods:

- `MoveTo`: This method moves the floor plan element to the new location specified by the `x` and `y` arguments
- `PrintCategory`: This method prints the value of the `Category` class attribute to the console output
- `PrintDescription`: This method prints the value of the `Description` class attribute to the console output
- `Draw`: This method prints the category, description, 2D location, width, and height of the floor plan element to the console output

The `FloorPlanElement` abstract class is located in the `General\FloorPlanElement.cs` file. All the classes that specialize floor plan elements will inherit from the `FloorPlanElement` class; therefore they will implement the `IFloorPlanElement` interface. These classes will be located in other C# source files that will have to use the `using` statement to let C# know that they want to use the `FloorPlanElement` class that is located in another namespace, that is, in the `Chapter7.General` namespace.

Using namespaces

We will create the following folders in the previously created `Build` folder. We won't add C# source files to all the folders in order to keep our example simple. However, we will imagine that we have a more complex project:

- `Areas`
- `Doors`
- `Fireplaces`
- `Rooms`
- `Stairs`
- `Walls`
- `Windows`

Now, we will add the `Room` abstract class, which inherits from the previously created `FloorPlanElement` abstract class, to a file named `Room.cs` within the `Build\Rooms` subfolder. The following lines show the code for the `Build\Rooms\Room.cs` C# source file that declares the `Room` class. The class just overrides the getter and setter methods for the `Category` property. The getter method returns the `"Room"` value. In addition, the class declares a constructor that just calls the base constructor:

```
namespace Chapter7.Build.Rooms
{
  using System;
  using General;

  public abstract class Room : FloorPlanElement
  {
    public override string Category
    {
      get { return "Room"; }
      set { throw new InvalidOperationException(); }
    }

    public Room(double x, double y, double width, double height,
IFloorPlanElement parent) : base(x, y, width, height, parent)
    {
    }
  }
}
```

If we use Visual Studio to create a new interface in the `Build\Rooms` folder, the IDE will automatically include a line with the `namespace` keyword, followed by the `Chapter7.Build.Rooms` name. The declaration of the interface will be enclosed in brackets after the line that declares the namespace. The IDE uses the folder names in which we will add the interface to automatically generate a default namespace name. In this case, the generated name is the initial namespace, that is, the solution name, the folder, and the subfolder, all of them separated by a dot (`.`), that is, `Chapter7.Build.Rooms`.

The new class inherits from the `FloorPlanElement` class that was declared in another namespace, that is, in the `Chapter7.General` namespace. We had to specify the class name from which the new abstract class inherits; therefore, we added the `using` statement, followed by `General`. This way, we don't need to specify the full qualifier for the `FloorPlanElement` class; we can reference it by just using its class name.

Bear in mind that the real namespace name is `Chapter7.General`. However, we are under the scope of the `Chapter7` namespace because we include the `using` line in the namespace declaration so that we don't need to include `Chapter7.` as a prefix. This way, we can just specify `General`. If we don't include the `using` statement, followed by the `General` namespace, we should use a full qualifier to reference the `FloorPlanElement` class, as shown in the following line that declares the `Room` abstract class:

```
public abstract class Room : General.FloorPlanElement
```

In addition, it will be necessary to add a full qualifier to reference the `IFloorPlanElement` interface in the constructor declaration:

```
public Room(double x, double y, double width, double height, General.
IFloorPlanElement parent) : base(x, y, width, height, parent)
```

If we decide to include the `using` statement before and outside the namespace declaration, we should use the following line:

```
using Chapter7.General;
```

Now, we will add a `SquareRoom` class, which inherits from the previously created `Room` abstract class, to a file named `SquareRoom.cs` within the `Build\Rooms` subfolder. The following is the code for the `Build\Rooms\SquareRoom.cs` C# source file that declares the `SquareRoom` class. The class just overrides the getter and setter methods for the `Description` property. The getter method returns the `"Square room"` value. The `SquareRoom` class represents a square room; therefore, it isn't necessary to specify both the width and height to create an instance of this type of room. The constructor uses the `width` value to specify the values for both width and height in the call to the base constructor:

```
namespace Chapter7.Build.Rooms
{
  using System;
  using General;

  class SquareRoom : Room
  {
    public override string Description
    {
      get { return "Square room"; }
      set { throw new InvalidOperationException(); }
    }

    public SquareRoom(double x, double y, double width,
IFloorPlanElement parent) : base(x, y, width, width, parent)
```

```
        {
        }
      }
    }
```

As occurred in the `Room` abstract class file, we added the `using` statement, followed by `General`. This way, we don't need to specify the full qualifier for the `IFloorPlanElement` interface. The `Room` class is declared in the same namespace in which we declare this new class; therefore, the `Room` class is under scope.

Now, we will add the `LShapedRoom` class, which also inherits from the `Room` abstract class, to a file named `LShapedRoom.cs` within the `Build\Rooms` subfolder. The following code shows the `Build\Rooms\LShapedRoom.cs` C# source file that declares the `SquareRoom` class. The class just overrides the getter and setter methods for the `Description` property. The getter method returns the `"L-Shaped room"` value. In addition, the class declares a constructor that just calls the base constructor:

```
namespace Chapter7.Build.Rooms
{
  using System;
  using General;

  public class LShapedRoom: Room
  {
    public override string Description
    {
      get { return "L-Shaped room"; }
      set { throw new InvalidOperationException(); }
    }

    public LShapedRoom(double x, double y, double width, double
height, IFloorPlanElement parent) : base(x, y, width, height, parent)
    {
    }
  }
}
```

Now, we will add the `SmallRoom` class that also inherits from the `Room` abstract class to a file named `SmallRoom.cs` within the `Build\Rooms` subfolder. The following code shows the `Build\Rooms\SmallRoom.cs` C# source file that declares the `SmallRoom` class. The class just overrides the getter and setter methods for the `Description` property. The getter method returns the `"Small room"` value. In addition, the class declares a constructor that just calls the base constructor:

```
namespace Chapter7.Build.Rooms
{
  using System;
  using General;

  public class SmallRoom : Room
  {
    public override string Description
    {
      get { return "Small room"; }
      set { throw new InvalidOperationException(); }
    }

    public SmallRoom(double x, double y, double width, double height,
IFloorPlanElement parent)
      : base(x, y, width, height, parent)
    {
    }
  }
}
```

Now, we will add the `Closet` class (which also inherits from the `Room` abstract class) to a file named `Closet.cs` within the `Build\Rooms` subfolder. The following is the code for the `Build\Rooms\Closet.cs` C# source file that declares the `Closet` class. The class just overrides the getter and setter methods for the `Description` property. The getter method returns the `"Closet"` value. In addition, the class declares a constructor that just calls the base constructor:

```
  namespace Chapter7.Build.Rooms
  {
    using System;
    using General;

    public class Closet : Room
    {
      public override string Description
      {
        get { return "Closet"; }
        set { throw new InvalidOperationException(); }
      }

      public Closet(double x, double y, double width, double height,
  IFloorPlanElement parent)
        : base(x, y, width, height, parent)
      {
      }
    }
  }
```

We will add the `Door` abstract class, which inherits from the previously created `FloorPlanElement` abstract class, to a file named `Door.cs` within the `Build\Doors` subfolder. The following is the code for the `Build\Doors\Door.cs` C# source file that declares a `Door` class. This class just overrides the getter and setter methods for the `Category` property. The getter method returns the `"Door"` value. In addition, the `Door` class declares a constructor that just calls the base constructor:

```
namespace Chapter7.Build.Doors
{
  using System;
  using General;

  public abstract class Door : FloorPlanElement
  {
    public override string Category
    {
      get { return "Door"; }
      set { throw new InvalidOperationException(); }
    }

    public Door(double x, double y, double width, double height,
IFloorPlanElement parent)
      : base(x, y, width, height, parent)
    {
    }
  }
}
```

The `Door` class is included in the `Chapter7.Build.Doors` namespace. The new class inherits from the `FloorPlanElement` class that was declared in another namespace, that is, in the `Chapter7.General` namespace.

Now, we will add the `EntryDoor` class, which inherits from the `Door` abstract class, to a file named `EntryDoor.cs` within the `Build\Doors` subfolder. The following is the code for the `Build\Doors\EntryDoor.cs` C# source file that declares the `EntryDoor` class. This class just overrides the getter and setter methods for the `Description` property. The getter method returns the `"Entry Door"` value. In addition, the class declares a constructor that just calls the base constructor:

```
namespace Chapter7.Build.Doors
{
  using System;
  using General;

  public class EntryDoor : Door
```

```
    {
      public override string Description
      {
        get { return "Entry Door"; }
        set { throw new InvalidOperationException(); }
      }

      public EntryDoor(double x, double y, double width, double height,
  IFloorPlanElement parent)
        : base(x, y, width, height, parent)
      {
      }
    }
  }
```

We will create the following subfolders in the previously created `Decorate` folder. Each of these subfolders will generate a namespace in the `Chapter7.Decorate` namespace and will include the following classes:

- `ArtAndDecor`
- `Countertops`
- `Electronics`
- `Flooring`
- `LightingAndFans`
- `PaintAndWalls`

We will create the following subfolders in the previously created `Furnish` folder. Each of these subfolders will generate a namespace in the `Chapter7.Furnish` namespace and will include classes. The `Bedroom` subfolder will include the additional subfolders:

- `Bathroom`
- `Bedroom`
- `DiningRoom`
- `Kitchen`
- `LaundryAndUtility`
- `LivingRoom`
- `Office`
- `OtherRooms`

We will create the following subfolders within the previously created `Furnish\Bedroom` subfolder:

- `Beds`
- `Dressers`
- `KidsBeds`
- `Mirrors`
- `NightTables`
- `Nursery`

We will add the `Bed` class, which inherits from the previously created `FloorPlanElement` abstract class, to a file named `Bed.cs` within the `Furnish\Bedroom\Beds` subfolder. The following is the code for the `Furnish\Bedroom\Beds\Bed.cs` C# source file that declares the `Bed` class. The class just overrides the getter and setter methods for the properties of `Category` and `Description`. The getter method for the `Category` property returns the `"Bed"` value and the getter method for `Description` returns `"Generic Bed"`. In addition, the class declares a constructor that just calls the base constructor:

```
namespace Chapter7.Furnish.Bedroom.Beds
{
  using System;
  using General;

  public class Bed : FloorPlanElement
  {
    public override string Category
    {
      get { return "Bed"; }
      set { throw new InvalidOperationException(); }
    }

    public override string Description
    {
      get { return "Generic Bed"; }
      set { throw new InvalidOperationException(); }
    }

    public Bed(double x, double y, double width, double height,
IFloorPlanElement parent)
```

```
        : base(x, y, width, height, parent)
      {
      }
    }
  }
```

The Bed class is included in the Chapter7.Furnish.Bedroom.Beds namespace. The new class inherits from the FloorPlanElement class. This class was declared in another namespace, that is, in the Chapter7.General namespace.

Now, we will add the FabricBed class to a file named FabricBed.cs within the Furnish\Bedroom\Beds subfolder. The FabricBed class inherits from the Bed class. The following is the code for the Furnish\Bedroom\Beds\FabricBed.cs C# source file that declares the FabricBed class. The class just overrides the getter and setter methods for the Description property. The getter method returns the "Fabric Bed" value. In addition, the class declares a constructor that just calls the base constructor:

```
namespace Chapter7.Furnish.Bedroom.Beds
{
  using System;
  using General;

  public class FabricBed : Bed
  {
    public override string Description
    {
      get { return "Fabric Bed"; }
      set { throw new InvalidOperationException(); }
    }

    public FabricBed(double x, double y, double width, double height,
IFloorPlanElement parent)
      : base(x, y, width, height, parent)
    {
    }
  }
}
```

We will create the following subfolders in the previously created `Landscape` folder. Each of these subfolders will generate a namespace in the `Chapter7.Landscape` namespace and will include the following classes:

- `AreasDefinition`
- `Materials`
- `TreesAndPlants`

Finally, we will create the following subfolders in the previously created `Outdoor` folder. Each of these subfolders will generate a namespace in the `Chapter7.Outdoor` namespace and will include the following classes:

- `Accessories`
- `Living`
- `Structures`

Working with namespace hierarchies in C#

Now, we will change the code for the `Program.cs` C# source file in the project's root folder, that is, the same folder that includes the `Build`, `Decorate`, `Furnish`, `General`, `Landscape`, and `Outdoor` subfolders. The following code uses many `using` directives to list all the namespaces to be used frequently. Then, the `Main` method works with instances of the classes defined in many different namespaces:

```
namespace Chapter7
{
  using System;
  using Build.Rooms;
  using Build.Doors;
  using Furnish.Bedroom.Beds;

  class Program
  {
    static void Main(string[] args)
    {
      var room1 = new SquareRoom(0, 0, 200, null);
      var door1 = new EntryDoor(100, 1, 50, 5, room1);
      var bedroom1 = new SquareRoom(100, 200, 180, null);
      var bed1 = new FabricBed(130, 230, 120, 110, bedroom1);
      room1.Draw();
      door1.Draw();
      bedroom1.Draw();
      bed1.Draw();
```

```
            Console.ReadLine();
        }
    }
}
```

The first line in the preceding code declares that we are working in the `Chapter7` namespace and using many `using` directives to make it simpler to access the classes declared in the specified namespaces:

- `Build.Rooms`: `Chapter7.Build.Rooms`
- `Build.Doors`: `Chapter7.Build.Doors`
- `Furnish.Bedroom.Beds`: `Chapter7.Bedroom.Beds`

This way, we can access the `SquareRoom`, `EntryDoor`, and `FabricBed` classes without any prefixes as if they were defined in the `Chapter7` namespace. The following lines show the output generated with the preceding code:

```
Room
Square room
X: 0, Y: 0. Width: 200, Height: 200.
Door
Entry Door
X: 100, Y: 1. Width: 50, Height: 5.
Room
Square room
X: 100, Y: 200. Width: 180, Height: 180.
Bed
Fabric Bed
X: 130, Y: 230. Width: 120, Height: 110.
```

In this case, we just need these three classes. Remember that we have included the `using` directives in the `Chapter7` namespace declaration. If we want to move the `using` directives before and outside the namespace declaration, we should use the following code that adds `Chapter7.` as a prefix to each namespace:

```
using Chapter7.Build.Rooms;
using Chapter7.Build.Doors;
using Chapter7.Furnish.Bedroom.Beds;
```

Another option is to remove the `using` directives and use full qualifiers for each of the required classes, that is, include the complete namespace name and a dot (`.`) as a prefix for each class. The following is the version of the code that removes the `using` directives and works with full qualifiers for each class:

```
namespace Chapter7
{
  using System;

  class Program
  {
    static void Main(string[] args)
    {
      var room1 = new Build.Rooms.SquareRoom(0, 0, 200, null);
      var door1 = new Build.Doors.EntryDoor(100, 1, 50, 5, room1);
      var bedroom1 = new Build.Rooms.SquareRoom(100, 200, 180, null);
      var bed1 = new Furnish.Bedroom.Beds.FabricBed(130, 230, 120,
110, bedroom1);
      room1.Draw();
      door1.Draw();
      bedroom1.Draw();
      bed1.Draw();

      Console.ReadLine();
    }
  }
}
```

Now, let's imagine that we have to code an application that has to draw the floor plans for 30 extremely complex houses that will be displayed in a 4K display. We will have to work with most of the elements defined in the different namespaces. We would like to have all the classes defined in the diverse namespaces accessed without prefixes. We would require the following `using` directives in the `Chapter7` namespace declaration:

```
using Build.Areas;
using Build.Doors;
using Build.Fireplaces;
using Build.Rooms;
using Build.Stairs;
using Build.Walls;
using Build.Windows;
using Decorate.ArtAndDecor;
using Decorate.Countertops;
using Decorate.Electronics;
using Decorate.Flooring;
using Decorate.LightingAndFans;
using Decorate.PaintAndWalls;
using Furnish.Bathroom;
using Furnish.Bedroom.Beds;
using Furnish.Bedroom.Dressers;
using Furnish.Bedroom.KidsBeds;
using Furnish.Bedroom.Mirrors;
using Furnish.Bedroom.NightTables;
using Furnish.Bedroom.Nursery;
using Furnish.DiningRoom;
using Furnish.Kitchen;
using Furnish.LaundryAndUtility;
using Furnish.LivingRoom;
using Furnish.Office;
using Furnish.OtherRooms;
using Landscape.AreasDefinition;
using Landscape.Materials;
using Landscape.TreesAndPlants;
using Outdoor.Accesories;
using Outdoor.Living;
using Outdoor.Structures;
```

Organizing object-oriented code in JavaScript

JavaScript was born as a scripting language that has grown up to become a language that creates entire apps. The usage of plain JavaScript without additional libraries doesn't provide a standardized mechanism to organize code in namespaces or modules.

We can easily organize our constructor functions with plain JavaScript, but in some cases, we can benefit from the usage of specialized libraries, such as `Require.js` (`http://www.requirejs.org/`), that provide a better mechanism to organize complex code in modules and solve the problem of dependencies and different ways of loading modules as well. In this case, we will organize our code using plain JavaScript without additional libraries.

Working with objects to organize code

We will create a base global object named `APP` to use it to define all the elements of our house floor plan layout application. Then, we will add the following properties to the base object to create a hierarchy of objects linked to this base object:

- `Build`
- `Decorate`
- `Furnish`
- `General`
- `Landscape`
- `Outdoor`

We will add the base constructor function in the `APP.General` object. The following code creates the `APP` object if it doesn't exist. It also defines the `General` property if it doesn't exist. Then, the code defines the `FloorPlanElement` constructor function and its prototype. We will use this constructor functions as the base of the prototype chain for other objects that will specialize the floor plan element:

```
var APP = APP || {};
APP.General = APP.General || {};
APP.General.FloorPlanElement = function() { };
APP.General.FloorPlanElement.prototype.category = "Undefined";
APP.General.FloorPlanElement.prototype.description = "Undefined";
APP.General.FloorPlanElement.prototype.x = 0;
APP.General.FloorPlanElement.prototype.y = 0;
APP.General.FloorPlanElement.prototype.width = 0;
APP.General.FloorPlanElement.prototype.height = 0;
APP.General.FloorPlanElement.prototype.parent = null;

APP.General.FloorPlanElement.prototype.initialize = function(x, y,
width, height, parent) {
  this.x = x;
  this.y = y;
  this.width = width;
  this.height = height;
  this.parent = parent;
}

APP.General.FloorPlanElement.prototype.moveTo = function(x, y) {
  this.x = x;
```

```
    this.y = y;
  }

  APP.General.FloorPlanElement.prototype.printCategory = function() {
    console.log(this.category);
  }

  APP.General.FloorPlanElement.prototype.printDescription = function() {
    console.log(this.description);
  }

  APP.General.FloorPlanElement.prototype.draw = function() {
    this.printCategory();
    this.printDescription();
    console.log("X: " + this.x +
      ", Y: " + this.y +
      ". Width: " + this.width +
      ", Height: " + this.height + ".");
  }
```

The `FloorPlanElement` prototype declares two properties that the subclasses will override: `category` and `description`. The prototype declares an `initialize` method that receives five arguments: `x`, `y`, `width`, `height`, and `parent`. The `initialize` method initializes the current instance properties with the same name with all the values received as arguments. This way, each constructor function in the prototype chain that ends up in the `FloorPlanElement` prototype will have a 2D location specified by `x` and `y`, `width` and `height` values, and a `parent` element. Each constructor function in the prototype chain will call the `initialize` method to initialize the instance.

In addition, the `FloorPlanElement` prototype declares the following methods:

- `moveTo`: This method moves the floor plan element to the new location specified by the `x` and `y` arguments
- `printCategory`: This method prints the value of the `category` property
- `printDescription`: This method prints the value of the `description` property
- `draw`: This method prints the category, description, 2D location, and width and height of the floor plan element

In this case, we will include all the constructor functions in the same JavaScript file. However, the same code can be moved to different JavaScript source files because whenever we declare a new group of related constructor functions, we will include the first few lines of the preceding code that creates the `APP` object if it doesn't exist and defines the property we use to generate the hierarchy of related constructor functions if it doesn't exist. However, if we separate code in different JavaScript source files, we have to make sure that we load them in the necessary order to avoid dependency problems. We won't add code to check for the existence of specific constructor functions, but we can easily define flags for this goal.

Declaring constructor functions within objects

The following code declares five constructor functions: `Room`, `SquareRoom`, `LShapedRoom`, `SmallRoom`, and `Closet`. The code defines these constructor functions as properties of the `APP.Build.Rooms` object:

```
var APP = APP || {};
APP.Build = APP.Build || {};
APP.Build.Rooms = APP.Build.Rooms || {};
APP.Build.Rooms.Room = function() { };
APP.Build.Rooms.Room.prototype = new APP.General.FloorPlanElement();
APP.Build.Rooms.Room.prototype.constructor = APP.Build.Rooms.Room;
APP.Build.Rooms.Room.prototype.category = "Room";

APP.Build.Rooms.SquareRoom = function(x, y, width, parent) {

  this.initialize(x, y, width, width, parent);
};
APP.Build.Rooms.SquareRoom.prototype = new APP.Build.Rooms.Room();
APP.Build.Rooms.SquareRoom.prototype.constructor = APP.Build.Rooms.
SquareRoom;
APP.Build.Rooms.SquareRoom.prototype.description = "Square room";

APP.Build.Rooms.LShapedRoom = function(x, y, width, height, parent) {
  this.initialize(x, y, width, height, parent);
};
APP.Build.Rooms.LShapedRoom.prototype = new APP.Build.Rooms.Room();
APP.Build.Rooms.LShapedRoom.prototype.constructor = APP.Build.Rooms.
LShapedRoom;
APP.Build.Rooms.LShapedRoom.prototype.description = "L-Shaped room";

APP.Build.Rooms.SmallRoom = function(x, y, width, height, parent) {
  this.initialize(x, y, width, height, parent);
};
APP.Build.Rooms.SmallRoom.prototype = new APP.Build.Rooms.Room();
APP.Build.Rooms.SmallRoom.prototype.constructor = APP.Build.Rooms.
SmallRoom;
APP.Build.Rooms.SmallRoom.prototype.description = "Small room";

APP.Build.Rooms.Closet = function(x, y, width, height, parent) {
  this.initialize(x, y, width, height, parent);
};
APP.Build.Rooms.Closet.prototype = new APP.Build.Rooms.Room();
APP.Build.Rooms.Closet.prototype.constructor = APP.Build.Rooms.Closet;
APP.Build.Rooms.Closet.prototype.description = "Closet";
```

The `APP.Build.Rooms.Room` constructor function specifies the previously defined `APP.General.FloorPlanElement` object as its prototype. The `APP.Build.Rooms.Room` prototype overrides the value of the `category` property with the `"Room"` value. The `App.Build.Rooms.SquareRoom` constructor function generates an object that represents the square room; therefore, it isn't necessary to specify both the width and height to create an instance of this type of room. The constructor function uses the `width` value to specify the values for both `width` and `height` in the call to the `initialize` method defined in the `APP.General.FloorPlanElement` prototype. Each constructor function that defines `APP.Build.Rooms.Room` as its prototype overrides the `description` property with an appropriate value.

The following code declares two constructor functions: `Door` and `EntryDoor`. The following code defines these constructor functions as properties of the `APP.Build.Doors` object:

```
  var APP = APP || {};
  APP.Build = APP.Build || {};
  APP.Build.Doors = APP.Build.Doors || {};
  APP.Build.Doors.Door = function() { };
  APP.Build.Doors.Door.prototype = new APP.General.FloorPlanElement();
  APP.Build.Doors.Door.prototype.constructor = APP.Build.Doors.Door;
  APP.Build.Doors.Door.prototype.category = "Door";

  APP.Build.Doors.EntryDoor = function(x, y, width, height, parent) {
    this.initialize(x, y, width, height, parent);
  };
  APP.Build.Doors.EntryDoor.prototype = new APP.Build.Doors.Door();
  APP.Build.Doors.EntryDoor.prototype.constructor = APP.Build.Doors.
  EntryDoor;
  APP.Build.Doors.EntryDoor.prototype.description = "Entry Door";
```

The `APP.Build.Doors.Door` constructor function specifies the previously defined `APP.General.FloorPlanElement` object as its prototype. The `APP.Build.Doors.Door` prototype overrides the value of the `category` property with the `"Door"` value. The `App.Build.Doors.Door` constructor function generates an object that represents an entry door; its prototype overrides the `description` property with the `"Entry Door"` value.

The following code declares two constructor functions: `Bed` and `FabricBed`. The code defines these constructor functions as properties of the `APP.Furnish.Bedroom.Beds` object:

```
var APP = APP || {};
APP.Furnish = APP.Furnish || {};
APP.Furnish.Bedroom = APP.Furnish.Bedroom || {};
APP.Furnish.Bedroom.Beds = APP.Furnish.Bedroom.Beds || {};
APP.Furnish.Bedroom.Beds.Bed = function() { };
APP.Furnish.Bedroom.Beds.Bed.prototype = new APP.General.
FloorPlanElement();
APP.Furnish.Bedroom.Beds.Bed.prototype.constructor = APP.Furnish.
Bedroom.Beds.Bed;
APP.Furnish.Bedroom.Beds.Bed.prototype.category = "Bed";
APP.Furnish.Bedroom.Beds.Bed.prototype.description = "Generic Bed";

APP.Furnish.Bedroom.Beds.FabricBed = function(x, y, width, height,
parent) {
  this.initialize(x, y, width, height, parent);
};
APP.Furnish.Bedroom.Beds.FabricBed.prototype = new APP.Furnish.
Bedroom.Beds.Bed();
APP.Furnish.Bedroom.Beds.FabricBed.prototype.constructor = APP.
Furnish.Bedroom.Beds.FabricBed;
APP.Furnish.Bedroom.Beds.FabricBed.prototype.description = "Fabric
Bed";
```

The `APP.Furnish.Bedroom.Beds.Bed` constructor function specifies the previously defined `APP.General.FloorPlanElement` object as its prototype. The `APP.Furnish.Bedroom.Beds.Bed` prototype overrides the value of both the `category` and `description` properties with the `"Bed"` and `"Generic Bed"` values. The `APP.Furnish.Bedroom.Beds.FabricBed` constructor function generates an object that represents a fabric bed; its prototype overrides the `description` property with the `"Fabric Bed"` value.

Working with nested objects that organize code

Now, we will write code that checks whether the previously defined objects that organized the code are defined. If they are defined, the following code creates objects with the previously defined constructor functions and calls the `draw` methods for all of the created objects:

```
if (!APP.Build.Rooms) {
  throw "Rooms objects not available.";
}
if (!APP.Build.Doors) {
  throw "Doors objects not available.";
}
if (!APP.Furnish.Bedroom.Beds) {
  throw "Beds objects not available.";
}

var room1 = new APP.Build.Rooms.SquareRoom(0, 0, 200, null);
var door1 = new APP.Build.Doors.EntryDoor(100, 1, 50, 5, room1);
var bedroom1 = new APP.Build.Rooms.SquareRoom(100, 200, 180, null);
var bed1 = new APP.Furnish.Bedroom.Beds.FabricBed(130, 230, 120, 110,
bedroom1);

room1.draw();
door1.draw();
bedroom1.draw();
bed1.draw();
```

The following lines show the output generated on the JavaScript console:

```
Room
Square room
X: 0, Y: 0. Width: 200, Height: 200.
Door
Entry Door
X: 100, Y: 1. Width: 50, Height: 5.
Room
Square room
X: 100, Y: 200. Width: 180, Height: 180.
Bed
Fabric Bed
X: 130, Y: 230. Width: 120, Height: 110.
```

Summary

In this chapter, you learned how to use all the features included in Python, C#, and JavaScript in order to organize complex object-oriented code. We took advantage of modules in Python, namespaces in C#, and nested objects in JavaScript. We organized multiple classes, interfaces, and constructor functions of a house floor plan layout application. If the basic features included in JavaScript to organize code aren't enough, we can use specialized libraries (such as the popular RequireJS).

Now that you have learned how to organize object-oriented code, we are ready to understand how to move forward to take advantage of all the things you learned so far in this book and use them in our real-world applications in Python, JavaScript, and C#, which is the topic of the next chapter.

8
Taking Full Advantage of Object-Oriented Programming

In this chapter, you will learn how to refactor existing code to take advantage of all the object-oriented programming techniques that you have learned so far. We will take advantage of all the different available features to refactor a piece of code and prepare it for future requirements in each of the three covered programming languages: Python, JavaScript, and C#. We will cover the following topics:

- Putting together all the pieces of the object-oriented puzzle
- Understanding the difference between writing object-oriented code from scratch and refactoring existing code
- Preparing object-oriented code for future requirements
- Refactoring existing code in Python
- Refactoring existing code in C#
- Refactoring existing code in JavaScript

Putting together all the pieces of the object-oriented puzzle

In *Chapter 1*, *Objects Everywhere*, you learned how to recognize objects from real-life situations. We understood that working with objects makes it easier to write code that is easier to understand and reuse. However, Python, C#, and JavaScript have different object-oriented approaches, and each programming language provides different features that allow you to generate and organize blueprints for objects. If we have the same goals for an application, we will end up with completely different object-oriented approaches in Python, C#, and JavaScript. It is not possible to use the same approach in these three languages.

In *Chapter 2*, *Classes and Instances*, you learned that in Python and C#, classes are the blueprints or building blocks that we can use to generate instances. Both the programming languages allow you to customize constructors and destructors. JavaScript has a different approach. We can easily create objects in JavaScript without any kind of blueprint. However, we can also take advantage of constructor functions and prototypes to group properties and methods that we can reuse to generate multiple objects using the same building blocks.

In *Chapter 3*, *Encapsulation of Data*, you learned about the different members of a class and how its different members are reflected in members of the instances generated from a class. We understood the possibility of protecting and hiding data and designed both mutable and immutable classes. Immutable classes are extremely useful when we work with concurrent code. Each programming language provides a different mechanism to protect and hide data. However, the three programming languages allow you to work with property getters and setters.

Python works with prefixes to indicate that we don't have to access specific members. C# is very strict and works with access modifiers to make it impossible for us to use members that we aren't supposed to access. In fact, if we try to access a member that isn't available for us to use, the code won't even compile. However, everything has a price, and C# adds an important amount of boilerplate code to provide these features.

In JavaScript, objects are extremely flexible and can easily mutate from the original form they acquired after we use a constructor function to create a new one. We can add members—such as methods and properties—on the fly. In fact, we can even change the prototype that is linked to an object and use all the members added to a prototype in the objects that we created before our changes. Bear in mind that JavaScript provides a mechanism that allows you to protect properties from being removed. We didn't take advantage of this feature in our examples.

In *Chapter 4*, *Inheritance and Specialization*, you learned about the different mechanisms provided by each programming language to specialize a blueprint. Python and C# work with inheritance; therefore, we work with classes that can become superclasses or base classes of a subclass or derived class. We worked with simple inheritance in both programming languages, performed methods and operators overriding, and took advantage of polymorphism. Also, we understood the power of overloading operators.

JavaScript works with prototype-based inheritance. We created objects that specialized behavior in this programming language. We also performed method overriding in JavaScript. We don't have to abuse large prototype chains in JavaScript because prototype-based inheritance can have a negative performance impact. We must take this into account, especially when we are used to taking full advantage of inheritance in other programming languages (Python and C#).

In *Chapter 5*, *Interfaces, Multiple Inheritance, and Composition*, you learned that C# works with interfaces in combination with classes. The only way to have multiple inheritance in C# is through the usage of interfaces. Interfaces are extremely useful, but they have a drawback, that is, they require us to write additional code. Luckily, there are tools included in all the modern IDEs. These IDEs allow you to easily and automatically generate an interface from a class without having to write all the code. We understood how we could use interfaces as the types required for arguments and that any instance of a class that implements the interface can be used as an argument.

Python allows you to work with multiple inheritance of classes; therefore, we can declare a class with more than one superclass. However, we should use multiple inheritances carefully to avoid generating a big mess. Python doesn't work with an interface, but works with a module that allows you to work with abstract base classes. It makes sense to use abstract base classes only in specific cases in Python.

We can work with composition in JavaScript to generate objects composed of many objects. This way, we can generate instances composed of objects created with diverse prototypes.

In *Chapter 6*, *Duck Typing and Generics*, you learned that both Python and JavaScript work with duck typing. We can add the necessary validation code to make sure that all the arguments have specific properties or belong to a specific type. However, the most common practice in both languages is to take advantage of duck typing.

C# uses interfaces in combination with generics to work with parametric polymorphism. We can declare classes that work with one or more constrained generic types. Generics are very important to maximize code reuse in C#.

In *Chapter 7*, *Organization of Object-Oriented Code*, you learned that Python allows you to easily organize source files in folders to define module hierarchies. C# works with namespaces, and we can easily match them with the location of all the source code files within folders in the project structure. We can declare a constructor function within objects in JavaScript and nest objects to organize code in JavaScript. However, if our code is complex and we want to use many files in different folders, we can use RequireJS, a popular code organization module.

Now, we will take our existing code and refactor it to take advantage of object-oriented programming.

Refactoring existing code in Python

Sometimes, we are extremely lucky and have the possibility to follow best practices as we kick off a project. If we start writing object-oriented code from scratch, we can take advantage of all the features that we have been using in our examples throughout the book. As the requirements evolve, we may need to further generalize or specialize all the blueprints. However, as we started our project with an object-oriented approach and organizing our code, it is easier to make adjustments to the code.

Most of the times, we aren't extremely lucky and have to work on projects that don't follow best practices. In the name of Agile, we generate pieces of code that perform similar tasks without a decent organization. Instead of following the same bad practices that generated error-prone, repetitive, and difficult to maintain code, we can use all the features provided by all the different IDEs and additional helper tools to refactor existing code and generate object-oriented code that promotes code reuse and allows you to reduce maintenance headaches.

For example, imagine that we have to develop an application that has to render 3D models on a 2D screen. The requirements specify that the first set of 3D models that we will have to render are a sphere and a cube. The application has to allow you to change parameters of a perspective camera in order to allow you to see a specific part of the 3D world rendered on the 2D screen (refer to *Figure 1* and *Figure 2*):

- The X, Y, and Z position
- The X, Y, and Z direction
- The X, Y, and Z up vector

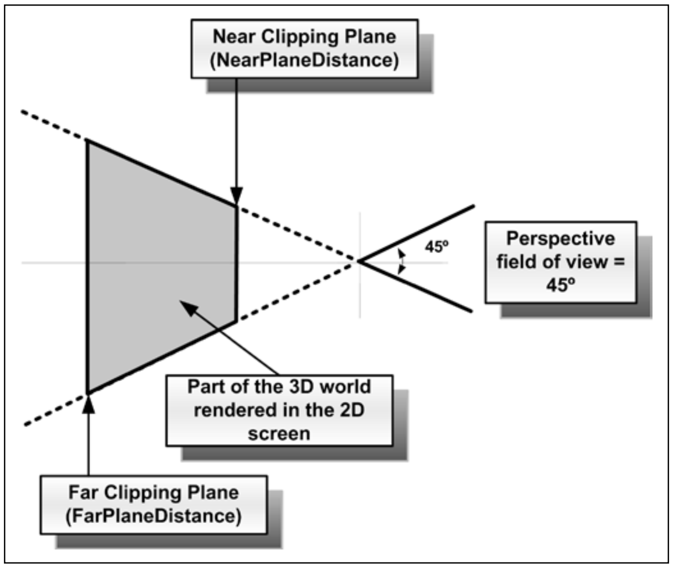

Near Clipping Plane
(NearPlaneDistance)
45°
Perspective field of view = 45°
Part of the 3D world rendered in the 2D screen
Far Clipping Plane
(FarPlaneDistance)

Figure 1

- **Perspective field of view in degrees**: This value determines the angle for the perspective camera's lens. A low value for this angle narrows the view. Thus, all the models will appear larger in the lens with a perspective field of a view of 45 degrees. A high value for this angle widens the view; therefore, all the models appear smaller in the visible part of the 3D world.

- **The near clipping plane**: The 3D region, which is visible on the 2D screen, is formed by a clipped pyramid called a frustum. This value controls the position of the plane that slices the top of the pyramid and determines the nearest part of the 3D world that the camera will render on the 2D screen. As the value is expressed taking into account the *Z* axis, it is a good idea to add code to check whether we are entering a valid value for this parameter.
- **The far clipping plane**: This value controls the position of the plane that slices the back of the pyramid and determines the more distant part of the 3D world that the camera will render on the 2D screen. The value is also expressed taking into account the *Z* axis; therefore, it is a good idea to add code to check whether we are entering a valid value for this parameter.

In addition, we can change the color of a directional light, that is, a light that casts light in a specific direction, which is similar to sunlight.

Let's start with the Python programming language. Here, we will be able to apply a similar procedure in C# or Java, considering the way we have been working with these languages in the previous chapters. Imagine that other developers started working on the project and generated a single Python source file with many functions that render a cube and a sphere. These functions receive all the necessary parameters to render each 3D figure, including the *X*, *Y*, and *Z* axes, determine the 3D figure's size, and configure the camera and the directional light:

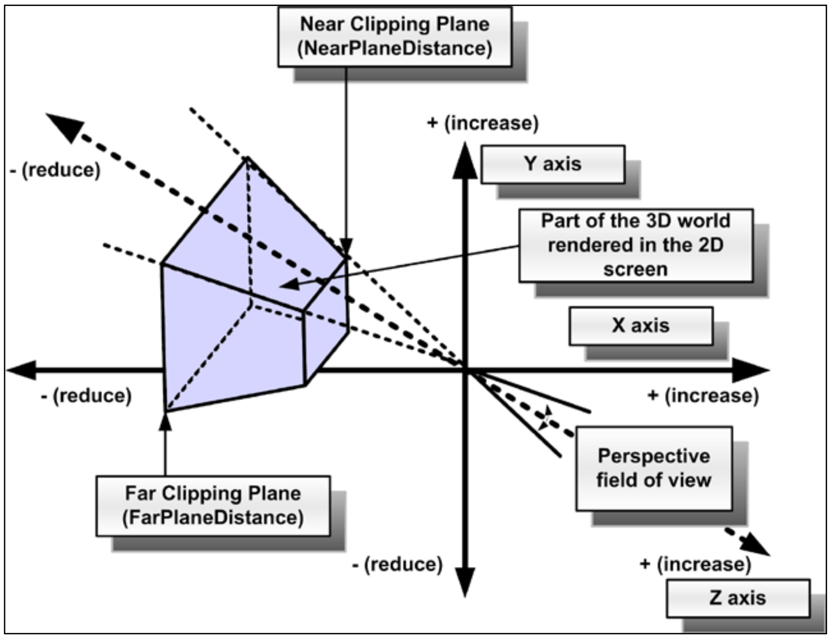

Near Clipping Plane (NearPlaneDistance)
+ (increase)
Y axis
- (reduce)
Part of the 3D world rendered in the 2D screen
X axis
- (reduce)
+ (increase)
Perspective field of view
Far Clipping Plane (FarPlaneDistance)
- (reduce)
+ (increase)
Z axis

Figure 2

The following code shows an example of the declaration of the function that renders a sphere named `render_sphere` and the function that renders a cube named `render_cube`:

```
def render_sphere(x, y, z, radius, camera_x, camera_y, camera_z,
camera_direction_x, camera_direction_y, camera_direction_z, camera_
vector_x, camera_vector_y, camera_vector_z, camera_perspective_
field_of_view, camera_near_clipping_plane, camera_far_clipping_plane,
directional_light_x, directional_light_y, directional_light_z,
directional_light_color):
    pass


def render_cube(x, y, z, edge_length, camera_x, camera_y, camera_z,
camera_direction_x, camera_direction_y, camera_direction_z, camera_
vector_x, camera_vector_y, camera_vector_z, camera_perspective_field_
of_view, camera_near_clipping_plane, camera_far_clipping_plane,
directional_light_x, directional_light_y, directional_light_z,
directional_light_color):
    pass
```

Each function requires a huge number of parameters. Let's imagine that we have requirements to add code in order to render additional shapes and add different types of cameras and lights. The preceding code can easily become a really big mess, repetitive, and difficult to maintain.

The first thing we can change is to work with a `Vector3D` class instead of working with separate *X*, *Y*, and *Z* values. Then, we can create a class for each of the following elements:

- `SceneElement`: This class represents a 3D element that is part of a scene and has a location specified with the `Vector3D` class. It is the base class for all the scene elements that require a location in the 3D space.
- `Light`: This is a subclass of `SceneElement` and represents a 3D light. It is the base class for all the lights.
- `DirectionalLight`: This is a subclass of `Light` and represents a directional light. It adds the property of `color`. The property setter makes sure that we cannot specify invalid values for the underlying attribute.
- `Camera`: This is a subclass of `SceneElement` and represents a 3D camera. It is the base class for all the cameras.

- `PerspectiveCamera`: This is a subclass of `Camera` and represents a perspective camera. It adds the `Vector3D` attributes: `direction` and `vector`. In addition, the class adds the `field_of_view`, `near_clipping_plane`, and `far_clipping_plane` properties. The property setters make sure that we cannot specify invalid values for all the underlying attributes.
- `Shape`: This is a subclass of `SceneElement` and represents a 3D shape that has a location specified with a `Vector3D` instance. It is the base class for all the 3D shapes and defines an empty `Render` method that receives a `Camera` instance.
- `Sphere`: This is a subclass of `Shape` that adds a `radius` property and overrides the `Render` method defined in its superclass to render a sphere.
- `Cube`: This is a subclass of `Shape` that adds an `edge_length` property and overrides the `Render` method defined in its superclass to render a sphere.
- `Scene`: This class represents the scene to be rendered.

The `Scene` class defines an `active_camera` attribute that holds the `Camera` instance. The `lights` attribute is a list of `Light` instances, whereas all the shapes attributes is a list of `Shape` instances that compose a scene. The `add_light` method adds `Light` to the `lights` list. The `add_shape` method adds `Shape` to the shapes list. Finally, the `render` method calls the render method for each of the `Shape` instances included in the `shapes` list and passes the `active_camera` attribute and the `lights` list as arguments.

The following code shows the Python code that defines the previously explained classes. In this case, we will use attributes; we haven't added any validation code to keep the code as simples as possible. In addition, this code doesn't really render any shapes because it would require a huge number of lines of code. Also, don't forget to organize the code as you learned in the previous chapter:

```
class Vector3D:
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z


class SceneElement:
    def __init__(self, location):
        self.location = location


class Light(SceneElement):
    def __init__(self, location):
        self.location = location


class DirectionalLight(Light):
    def __init__(self, location, color):
        super().__init__(location)
        self.color = color


class Camera(SceneElement):
    def __init__(self, location):
        super().__init__(location)


class PerspectiveCamera(Camera):
    def __init__(self, location, direction, vector, field_of_view,
near_clipping_plane, far_clipping_plane):
        super().__init__(location)
        self.direction = direction
        self.vector = vector
        self.field_of_view = field_of_view
        self.near_clipping_plane = near_clipping_plane
        self.far_clipping_plane = far_clipping_plane

class Shape(SceneElement):
    def __init__(self, location):
        super().__init__(location)

    def render(self, camera, lights):
        pass


class Sphere(Shape):
    def __init__(self, location, radius):
        super().__init__(location)
        self.radius = radius

    def render(self, camera, lights):
        print("Rendering a sphere.")

class Cube(Shape):
    def __init__(self, location, edge_length):
        super().__init__(location)
```

```
            self.edge_length = edge_length

        def render(self, camera, lights):
            print("Rendering a cube.")

    class Scene:
        def __init__(self, initial_camera):
            self.active_camera = initial_camera
            self.shapes = []
            self.lights = []

        def add_light(self, light):
            self.lights.append(light)

        def add_shape(self, shape):
            self.shapes.append(shape)

        def render(self):
            for shape in self.shapes:
                shape.render(self.active_camera, self.lights)
```

After we create the previously shown classes, we can enter the following code in the `__main__` method:

```
    if __name__ == '__main__':
        camera = PerspectiveCamera(Vector3D(30, 30, 30), Vector3D(50, 0,
    0), Vector3D(4, 5, 2), 90, 20, 40)
        sphere = Sphere(Vector3D(20, 20, 20), 8)
        cube = Cube(Vector3D(10, 10, 10), 5)
        light = DirectionalLight(Vector3D(2, 2, 5), 235)
        scene = Scene(camera)
        scene.add_shape(sphere)
        scene.add_shape(cube)
        scene.add_light(light)
        scene.render()
```

The preceding code is very easy to understand and read. First, we created `PerspectiveCamera` with all the necessary parameters. Then, we created two shapes: a `Sphere` and a `Cube`. Finally, we created `DirectionalLight` with all the necessary parameters and `Scene` with the previously created `PerspectiveCamera` as the initial camera. Then, we added all the shapes and the light to the scene and called the render method to render the scene.

Now, compare the previous code with the following main method that calls the `render_sphere` and `render_cube` functions with more than a dozen parameters:

```
if __name__ == '__main__':
    render_sphere(20, 20, 20, 8, 30, 30, 30, 50, 0, 0, 4, 5, 2, 90, 
20, 40, 2, 2, 5, 235)
    render_cube(10, 10, 10, 5, 30, 30, 30, 50, 0, 0, 4, 5, 2, 90, 20, 
40, 2, 2, 5, 235)
```

The object-oriented version makes it easier to add parameters to any scene element. In addition, we can add the necessary light types, camera types, and shapes by specializing all the base classes. Whenever we find a behavior that is repeated in subclasses, we can generalize the code and move it to its base class. This way, we can reuse code and make future extensions to the application easier to code.

Refactoring existing code in C#

The following code shows an example of the declaration of the method that renders a sphere named `RenderSphere` and the method that renders a cube named `RenderCube` in C#:

```
public static void RenderSphere(
  int x, int y, int z, int radius,
  int cameraX, int cameraY, int cameraZ,
  int cameraDirectionX, int cameraDirectionY, int cameraDirectionZ,
  int cameraVectorX, int cameraVectorY, int cameraVvectorZ,
  int cameraPerspectiveFieldOfView,
  int cameraNearClippingPlane,
  int cameraFarClippingPlane,
  int directionalLightX, int directionalLightY, int directionalLightZ,
  int directionalLightColor)
{
}

public static void RenderCube(
  int x, int y, int z, int edgeLength,
  int cameraX, int cameraY, int cameraZ,
  int cameraDirectionX, int cameraDirectionY, int cameraDirectionZ,
  int cameraVectorX, int cameraVectorY, int cameraVvectorZ,
  int cameraPerspectiveFieldOfView,
  int cameraNearClippingPlane,
  int cameraFarClippingPlane,
  int directionalLightX, int directionalLightY, int directionalLightZ,
  int directionalLightColor)
{
}
```

Each function requires a huge number of parameters. Let's imagine that we have requirements to add code in order to render additional shapes and add different types of cameras and lights. The code can easily become a really big mess, repetitive, and difficult to maintain.

In *Chapter 3*, *Encapsulation of Data*, we worked with both mutable and immutable versions of the `Vector3D` class. Then, you learned how to overload operators in C#. The first thing we can change is to work with the `Vector3D` class instead of working with separate *X*, *Y*, and *Z* values. The following code shows a simple `Vector3D` class that uses auto-implemented properties. The code doesn't overload operators because we want to keep the example extremely simple:

```
public class Vector3D
{
  public int X { get; set; }
  public int Y { get; set; }
  public int Z { get; set; }

  public Vector3D(int x, int y, int z)
  {
    this.X = x;
    this.Y = y;
    this.Z = z;
  }
}
```

We will create a simple interface named `ISceneElement` to specify all the requirements for scene elements:

```
public interface ISceneElement
{
  Vector3D Location { get; set; }
}
```

The following code declares an abstract `SceneElement` class that implements the previously defined `ISceneElement` interface. The `SceneElement` class represents a 3D element. This element is part of a scene and has a location specified with `Vector3D`. It is the base class for all the scene elements that require a location in the 3D space:

```
public abstract class SceneElement : ISceneElement
{
  public Vector3D Location { get; set; }

  public SceneElement(Vector3D location)
  {
    this.Location = location;
  }
}
```

The following code declares another abstract class named `Light`. This is a subclass of the previously defined `SceneElement` class. This class represents a 3D light and is the base class for all the lights:

```
public abstract class Light : SceneElement
{
  public Light(Vector3D location)
    : base(location)
  {

  }
}
```

The following code declares a subclass of `Light` named `DirectionalLight`. The class represents a directional light and adds a `Color` property. In this case, we don't add validations for all the property setters just to make the example simple. However, we already know how to do it:

```
public class DirectionalLight : Light
{
  public int Color { get; set; }

  public DirectionalLight(Vector3D location, int color)
    : base(location)
  {
    this.Color = color;
  }
}
```

The following code declares an abstract class named `Camera` that inherits from `SceneElement`. The class represents a 3D camera. It is the base class for all the cameras:

```
public abstract class Camera : SceneElement
{
  public Camera(Vector3D location)
    : base(location)
  {
  }
}
```

The following code declares a subclass of `Camera` named `PerspectiveCamera`. The class represents a perspective camera and adds the `Direction` and `Vector` auto-implemented `Vector3D` properties. In addition, the class adds the `FieldOfView`, `NearClippingPlane`, and `FarClippingPlane` auto-implemented properties:

```
public class PerspectiveCamera : Camera
{
  public Vector3D Direction { get; set; }
  public Vector3D Vector { get; set; }
  public int FieldOfView { get; set; }
  public int NearClippingPlane { get; set; }
  public int FarClippingPlane { get; set; }

  public PerspectiveCamera(Vector3D location, Vector3D direction,
Vector3D vector, int fieldOfView, int nearClippingPlane, int
farClippingPlane)
    : base(location)
  {
    this.Direction = direction;
    this.Vector = vector;
    this.FieldOfView = fieldOfView;
    this.NearClippingPlane = nearClippingPlane;
    this.FarClippingPlane = farClippingPlane;
  }
}
```

The following code declares an abstract class named `Shape` that inherits from `SceneElement`. The class represents a 3D shape and is the base class for all the 3D shapes. The class defines an abstract `Render` method that receives the `Camera` instance and a list of `Light` instances:

```
public abstract class Shape: SceneElement
{
  public Shape(Vector3D location)
    : base(location)
  {
  }

  public abstract void Render(Camera camera, List<Light> lights);
}
```

The following code declares a `Sphere` class, a subclass of `Shape` that adds a `Radius` auto-implemented property and overrides the `Render` method defined in its abstract superclass to render a sphere:

```
public class Sphere : Shape
{
  public int Radius { get; set; }

  public Sphere(Vector3D location, int radius)
    : base(location)
  {
    this.Radius = radius;
  }

  public override void Render(Camera camera, List<Light> lights)
  {
    Console.WriteLine("Rendering a sphere.");
  }
}
```

The following code declares a `Cube` class, a subclass of `Shape` that adds an `EdgeLength` auto-implemented property and overrides the `Render` method defined in its abstract superclass to render a cube:

```
public class Cube : Shape
{
  public int EdgeLength { get; set; }

  public Cube(Vector3D location, int edgeLength)
    : base(location)
  {
    this.EdgeLength = edgeLength;
  }

  public override void Render(Camera camera, List<Light> lights)
  {
    Console.WriteLine("Rendering a cube.");
  }
}
```

Finally, the following code declares the `Scene` class that represents the scene to be rendered. The class defines an `_activeCamera` protected field that holds the `Camera` instance. The `_lights` protected field is a list of `Light` instances, and the `_shapes` protected field is a list of `Shape` instances that compose a scene. The `AddLight` method adds `Light` to the `_lights` list. The `AddShape` method adds `Shape` to the `_shapes` list. Finally, the `render` method calls the render method for each of the `Shape` instances included in the `_shapes` list and passes `_activeCamera` and `lights` list as arguments:

```
public class Scene
{
  protected List<Light> _lights;
  protected List<Shape> _shapes;
  protected Camera _activeCamera;

  public Scene(Camera initialCamera)
  {
    this._activeCamera = initialCamera;
    this._shapes = new List<Shape>();
    this._lights = new List<Light>();
  }

  public void AddLight(Light light)
  {
    this._lights.Add(light);
  }

  public void AddShape(Shape shape)
  {
    this._shapes.Add(shape);
  }

  public void Render()
  {
    foreach (var shape in this._shapes)
    {
      shape.Render(this._activeCamera, this._lights);
    }
  }
}
```

After we create the previously shown classes, we can enter the following code in the `Main` method of a Windows console application:

```
var camera = new PerspectiveCamera(
  new Vector3D(30, 30, 30),
  new Vector3D(50, 0, 0),
  new Vector3D(4, 5, 2),
  90, 20, 40);
var sphere = new Sphere(new Vector3D(20, 20, 20), 8);
var cube = new Cube(new Vector3D(10, 10, 10), 5);
var light = new DirectionalLight(new Vector3D(2, 2, 5), 235);
var scene = new Scene(camera);
scene.AddShape(sphere);
scene.AddShape(cube);
scene.AddLight(light);
scene.Render();

Console.ReadLine();
```

The preceding code is very easy to understand and read. First, we created a `PerspectiveCamera` instance with all the necessary parameters. Then we created two shapes: a `Sphere` and a `Cube`. Finally, we created `DirectionalLight` with all the necessary parameters and a `Scene` with the previously created `PerspectiveCamera` as the initial camera. Then, we added all the shapes and the light to the scene and called the `Render` method in order to render the scene.

Now, compare the previous code with the following code that call the `RenderSphere` and `RenderCube` methods with more than a dozen parameters:

```
RenderSphere(
  20, 20, 20,
  8, 30, 30,
  30, 50, 0,
  0, 4, 5,
  2, 90, 20,
  40, 2, 2,
  5, 235);
RenderCube(
  10, 10, 10,
  5, 30, 30,
  30, 50, 0,
  0, 4, 5,
  2, 90, 20,
  40, 2, 2,
  5, 235);
```

The object-oriented version requires a higher number of lines of code. However, it is easier to understand and expand based on future requirements. If you need to add a new type of light, a new shape, or a new type of camera, you know where to add the pieces of code, which classes to create, and the methods to change.

Refactoring existing code in JavaScript

The following code shows an example of the declaration of the function that renders a sphere named `renderSphere` and the function that renders a cube named `renderCube` in JavaScript:

```
function renderSphere(
  x, y, z, radius,
  cameraX, cameraY, cameraZ,
  cameraDirectionX, cameraDirectionY, cameraDirectionZ,
  cameraVectorX, cameraVectorY, cameraVvectorZ,
  cameraPerspectiveFieldOfView,
  cameraNearClippingPlane,
  cameraFarClippingPlane,
  directionalLightX, directionalLightY, directionalLightZ,
  directionalLightColor)
{

}

function renderCube(
  x, y, z, edgeLength,
  cameraX, cameraY, cameraZ,
  cameraDirectionX, cameraDirectionY, cameraDirectionZ,
  cameraVectorX, cameraVectorY, cameraVvectorZ,
  cameraPerspectiveFieldOfView,
  cameraNearClippingPlane,
  cameraFarClippingPlane,
  directionalLightX, directionalLightY, directionalLightZ,
  directionalLightColor)
{

}
```

Each function requires a huge number of parameters. Let's imagine that we have requirements to add code in order to render additional shapes and add different types of cameras and lights. The code can easily become a really big mess, repetitive, and difficult to maintain.

The first thing we can change is to define a `Vector3D` constructor function that provides the `x`, `y`, and `z` properties instead of working with separate *X*, *Y*, and *Z* values. The following code shows a very simple `APP.Math.Vector3D` constructor function:

```
var APP = APP || {};
APP.Math = APP.Math || {};
APP.Math.Vector3D = function(x, y, z) {
  this.x = x;
  this.y = y;
  this.z = z;
}
```

The following code declares an `APP.Scene.DirectionalLight` constructor function. This function generates objects that will represent a directional light. The constructor function defines the properties of `location` and `color`. We will use an `APP.Math.Vector3D` object for `location`:

```
var APP = APP || {};
APP.Scene = APP.Scene || {};
APP.Scene.DirectionalLight = function(location, color) {
  this.location = location;
  this.color = color;
}
```

The following code declares an `APP.Scene.PerspectiveCamera` constructor function. This function generates objects that represent a perspective camera. The constructor function defines the `location`, `direction`, and `Vector` properties that will hold the `APP.Math.Vector3D` object. In addition, this code declares and initializes the `fieldOfView`, `nearClippingPlane`, and `farClippingPlane` properties:

```
var APP = APP || {};
APP.Scene = APP.Scene || {};
APP.Scene.PerspectiveCamera = function(location, direction, vector,
fieldOfView, nearClippingPlane, farClippingPlane) {
  this.location = location;
  this.direction = direction;
  this.vector = vector;
  this.fieldOfView = fieldOfView;
  this.nearClippingPlane = nearClippingPlane;
  this.farClippingPlane = farClippingPlane;
}
```

The following code declares an `APP.Shape.Sphere` constructor function that receives `location` and `radius`. The code declares the location and radius properties with all its values received as arguments. The location property will hold an `APP.Math.Vector3D` object. The prototype defines a `render` method that prints a line, which simulates that it will render a sphere:

```
var APP = APP || {};
APP.Shape = APP.Shape || {};
APP.Shape.Sphere = function(location, radius) {
  this.location = location;
  this.radius = radius;
}
APP.Shape.Sphere.prototype.render = function(camera, lights) {
  console.log("Rendering a sphere");
}
```

The following code declares an `APP.Shape.Cube` constructor function that receives `location` and `edgeLength`. The code declares the location and `edgeLength` properties with all its values received as arguments. The location property will hold the `APP.Math.Vector3D` object. The prototype defines the `render` method that prints a line simulating that it will render a cube:

```
var APP = APP || {};
APP.Shape = APP.Shape || {};
APP.Shape.Cube = function(location, edgeLength) {
  this.location = location;
  this.edgeLength = edgeLength;
}
APP.Shape.Cube.prototype.render = function(camera, lights) {
  console.log("Rendering a cube");
}
```

Finally, the following code declares the `APP.Scene.Scene` constructor function. This function generates objects that represent the scene to be rendered. The constructor function receives an `initialCamera` argument that the code assigns to the `activeCamera` property. The `lights` property is an array that will hold `APP.Scene.DirectionalLight` objects. The `shapes` property is an array that will hold `APP.Shape.Sphere` or `APP.Shape.Cube` objects that compose a scene. The `addLight` method adds an `APP.Scene.DirectionalLight` object to the `lights` array. The `addShape` method adds either an `APP.Shape.Sphere` object or an `APP.Shape.Cube` object to the `shapes` array. Finally, the `render` method calls the render method for each of the elements in the `shape` array `Shape` instances included in the `_shapes` list and passes `_activeCamera` and the `lights` list as arguments:

```
var APP = APP || {};
```

```
APP.Scene = APP.Scene || {};
APP.Scene.Scene = function(initialCamera) {
  this.activeCamera = initialCamera;
  this.shapes = [];
  this.lights = [];
}
APP.Scene.Scene.prototype.addLight = function(light) {
  this.lights.push(light);
}
APP.Scene.Scene.prototype.addShape = function(shape) {
  this.shapes.push(shape);
}
APP.Scene.Scene.prototype.render = function() {
  this.shapes.forEach(function (shape) { shape.render(this.
activeCamera, this.lights); });
}
```

After we create the previously shown constructor functions and their prototypes, we can enter the following code on a JavaScript console:

```
var camera = new APP.Scene.PerspectiveCamera(
  new APP.Math.Vector3D(30, 30, 30),
  new APP.Math.Vector3D(50, 0, 0),
  new APP.Math.Vector3D(4, 5, 2),
  90, 20, 40);
var sphere = new APP.Shape.Sphere(new APP.Math.Vector3D(20, 20, 20),
8);
var cube = new APP.Shape.Cube(new APP.Math.Vector3D(10, 10, 10), 5);
var light = new APP.Scene.DirectionalLight(new APP.Math.Vector3D(2, 2,
5), 235);
var scene = new APP.Scene.Scene(camera);
scene.addShape(sphere);
scene.addShape(cube);
scene.addLight(light);
scene.render();
```

The preceding code is very easy to understand and read. We created an `APP.Scene.PerspectiveCamera` object with all the necessary parameters. Then, we created two shapes: `APP.Shape.Sphere` and `APP.Shape.Cube`. Finally, we created `APP.Scene.DirectionalLight` with all the necessary parameters and `APP.Scene.Scene` with the previously created `APP.Scene.PerspectiveCamera` as the initial camera. Then, we added all the shapes and the light to the scene and called the `render` method in order to render a scene.

Now, compare the previous code with the following code that calls the `renderSphere` and `renderCube` functions with more than a dozen parameters:

```
renderSphere(
  20, 20, 20,
  8, 30, 30,
  30, 50, 0,
  0, 4, 5,
  2, 90, 20,
  40, 2, 2,
  5, 235);
renderCube(
  10, 10, 10,
  5, 30, 30,
  30, 50, 0,
  0, 4, 5,
  2, 90, 20,
  40, 2, 2,
  5, 235);
```

The object-oriented version requires a higher number of lines of code. However, it is easier to understand and expand based on future requirements. We haven't included any kind of validations in order to keep the sample code as simple as possible and focus on the refactoring process. However, it is easy to make all the necessary changes to add the necessary validations as you learned in the previous chapters.

Summary

In this chapter, you learned how to use all the features included in Python, C#, and JavaScript in order to write simple and complex object-oriented code. We can even refactor existing code to take advantage of object-oriented programming in order to prepare the code for future requirements, reduce maintenance costs, and maximize code reuse.

Once you start working with object-oriented code and follow its best practices, it is difficult to stop writing code that works with objects. Objects are everywhere in real-life situations; therefore, it makes sense to code plenty of objects.

Now that you have learned to write object-oriented code, you are ready to use everything you learned in real-life applications that will not only rock, but also maximize code reuse and simplify maintenance.

Module 3

Object-Oriented JavaScript - Second Edition

Learn everything you need to know about OOJS in this comprehensive guide

1 Object-oriented JavaScript

Ever since the early days of the Web, there has been a need for more dynamic and responsive interfaces. While it's OK to read static HTML pages of text and even better when they are beautifully presented with the help of CSS, it's much more fun to engage with applications in our browsers, such as e-mail, calendars, banking, shopping, drawing, playing games, and text editing. All that is possible thanks to JavaScript, the programming language of the Web. JavaScript started with simple one-liners embedded in HTML, but is now used in much more sophisticated ways. Developers leverage the object-oriented nature of the language to build scalable code architectures made up of reusable pieces.

If you look at the past and present buzzwords in web development—DHTML, Ajax, Web 2.0, HTML5—they all essentially mean HTML, CSS, and JavaScript. HTML for *content*, CSS for *presentation*, and JavaScript for *behavior*. In other words, JavaScript is the glue that makes everything work together so that we can build rich web applications.

But that's not all, JavaScript can be used for more than just the Web.

JavaScript programs run inside a host environment. The web browser is the most common environment, but it's not the only one. Using JavaScript, you can create all kinds of widgets, application extensions, and other pieces of software, as you'll see in a bit. Taking the time to learn JavaScript is a smart investment; you learn one language and can then write all kinds of different applications running on multiple platforms, including mobile and server-side applications. These days, it's safe to say that JavaScript is everywhere.

This book starts from zero, and does not assume any prior programming knowledge other than some basic understanding of HTML. Although there is one chapter dedicated to the web browser environment, the rest of the book is about JavaScript in general, so it's applicable to all environments.

Let's start with the following:

- A brief introduction to the story behind JavaScript
- The basic concepts you'll encounter in discussions on object-oriented programming

A bit of history

Initially, the Web was not much more than just a number of scientific publications in the form of static HTML documents connected together with hyperlinks. Believe it or not, there was a time when there was no way to put an image in a page. But that soon changed. As the Web grew in popularity and size, the webmasters who were creating HTML pages felt they needed something more. They wanted to create richer user interactions, mainly driven by the desire to save server roundtrips for simple tasks such as form validation. Two options came up: Java applets and LiveScript, a language conceived by Brendan Eich at Netscape in 1995 and later included in the Netscape 2.0 browser under the name of JavaScript.

The applets didn't quite catch on, but JavaScript did. The ability to use short code snippets embedded in HTML documents and alter otherwise static elements of a web page was embraced by the webmaster community. Soon, the competing browser vendor Microsoft shipped Internet Explorer (IE) 3.0 with JScript, which was a reverse engineered version of JavaScript plus some IE-specific features. Eventually, there was an effort to standardize the various implementations of the language, and this is how ECMAScript was born. **ECMA** (**European Computer Manufacturers Association**) created the standard called ECMA-262, which describes the core parts of the JavaScript programming language without browser and web page-specific features.

You can think of JavaScript as a term that encompasses three pieces:

- **ECMAScript**—the core language—variables, functions, loops, and so on. This part is independent of the browser and this language can be used in many other environments.
- **Document Object Model** (**DOM**), which provides ways to work with HTML and XML documents. Initially, JavaScript provided limited access to what's scriptable on the page, mainly forms, links, and images. Later it was expanded to make all elements scriptable. This lead to the creation of the DOM standard by the **World Wide Web Consortium** (**W3C**) as a language-independent (no longer tied to JavaScript) way to manipulate structured documents.

- **Browser Object Model** (**BOM**), which is a set of objects related to the browser environment and was never part of any standard until HTML5 started standardizing some of the common objects that exist across browsers.

While there is one chapter in the book dedicated to the browser, the DOM, and the BOM, most of the book describes the core language and teaches you skills you can use in any environment where JavaScript programs run.

Browser wars and renaissance

For better or for worse, JavaScript's instant popularity happened during the period of the Browser Wars I (approximately 1996 to 2001). Those were the times during the initial Internet boom when the two major browser vendors—Netscape and Microsoft—were competing for market share. Both were constantly adding more bells and whistles to their browsers and their versions of JavaScript, DOM, and BOM, which naturally led to many inconsistencies. While adding more features, the browser vendors were falling behind on providing proper development and debugging tools and adequate documentation. Often, development was a pain; you would write a script while testing in one browser, and once you're done with development, you test in the other browser, only to find that your script simply fails for no apparent reason and the best you can get is a cryptic error message like "Operation aborted".

Inconsistent implementations, missing documentation, and no appropriate tools painted JavaScript in such a light that many programmers simply refused to bother with it.

On the other hand, developers who did try to experiment with JavaScript got a little carried away adding too many special effects to their pages without much regard of how usable the end results were. Developers were eager to make use of every new possibility the browsers provided and ended up "enhancing" their web pages with things like animations in the status bar, flashing colors, blinking texts, objects stalking your mouse cursor, and many other "innovations" that actually hurt the user experience. These various ways to abuse JavaScript are now mostly gone, but they were one of the reasons why the language got some bad reputation. Many "serious" programmers dismissed JavaScript as nothing but a toy for designers to play around with, and dismissed it as a language unsuitable for serious applications. The JavaScript backlash caused some web projects to completely ban any client-side programming and trust only their predictable and tightly controlled server. And really, why would you double the time to deliver a finished product and then spend additional time debugging problems with the different browsers?

Everything changed in the years following the end of the Browser Wars I. A number of events reshaped the web development landscape in a positive way. Some of them are given as follows:

- Microsoft won the war with the introduction of IE6, the best browser at the time, and for many years they stopped developing Internet Explorer. This allowed time for other browsers to catch up and even surpass IE's capabilities.
- The movement for web standards was embraced by developers and browser vendors alike. Naturally, developers didn't like having to code everything two (or more) times to account for browsers' differences; therefore, they liked the idea of having agreed-upon standards that everyone would follow.
- Developers and technologies matured and more people started caring about things like usability, progressive enhancement techniques, and accessibility. Tools such as Firebug made developers much more productive and the development less of a pain.

In this healthier environment, developers started finding out new and better ways to use the instruments that were already available. After the public release of applications such as Gmail and Google Maps, which were rich on client-side programming, it became clear that JavaScript is a mature, unique in certain ways, and powerful prototypal object-oriented language. The best example of its rediscovery was the wide adoption of the functionality provided by the `XMLHttpRequest` object, which was once an IE-only innovation, but was then implemented by most other browsers. `XMLHttpRequest` allows JavaScript to make HTTP requests and get fresh content from the server in order to update some parts of a page without a full page reload. Due to the wide use of `XMLHttpRequest`, a new breed of desktop-like web applications, dubbed *Ajax* applications, was born.

The present

An interesting thing about JavaScript is that it always runs inside a *host environment*. The web browser is just one of the available hosts. JavaScript can also run on the server, on the desktop, and on mobile devices. Today, you can use JavaScript to do all of the following:

- Create rich and powerful web applications (the kind of applications that run inside the web browser). Additions to HTML5 such as application cache, client-side storage, and databases make browser programming more and more powerful for both online and offline applications.
- Write server-side code using .NET or `Node.js`, as well as code that can run using Rhino (a JavaScript engine written in Java).

- Make mobile applications; you can create apps for iPhone, Android, and other phones and tablets entirely in JavaScript using PhoneGap or Titanium. Additionally, apps for Firefox OS for mobile phones are entirely in JavaScript, HTML, and CSS.
- Create rich media applications (Flash, Flex) using ActionScript, which is based on ECMAScript.
- Write command-line tools and scripts that automate administrative tasks on your desktop using Windows Scripting Host or WebKit's JavaScript Core available on all Macs.
- Write extensions and plugins for a plethora of desktop applications, such as Dreamweaver, Photoshop, and most other browsers.
- Create cross operating system desktop applications using Mozilla's XULRunner or Adobe Air.
- Create desktop widgets using Yahoo! widgets or Mac Dashboard widgets. Interestingly, Yahoo! widgets can also run on your TV.

This is by no means an exhaustive list. JavaScript started inside web pages, but today it's safe to say it is practically everywhere. In addition, browser vendors now use speed as a competitive advantage and are racing to create the fastest JavaScript engines, which is great for both users and developers and opens doors for even more powerful uses of JavaScript in new areas such as image, audio, and video processing, and games development.

The future

We can only speculate what the future will be, but it's quite certain that it will include JavaScript. For quite some time, JavaScript may have been underestimated and underused (or maybe overused in the wrong ways), but every day we witness new applications of the language in much more interesting and creative ways. It all started with simple one liners, often embedded in HTML tag attributes (such as `onclick)`. Nowadays, developers ship sophisticated, well designed and architected, and extensible applications and libraries, often supporting multiple platforms with a single codebase. JavaScript is indeed taken seriously and developers are starting to rediscover and enjoy its unique features more and more.

Once listed in the "nice-to-have" sections of job postings, today, the knowledge of JavaScript is often a deciding factor when it comes to hiring web developers. Common job interview questions you can hear today include: "Is JavaScript an object-oriented language? Good. Now how do you implement inheritance in JavaScript?" After reading this book, you'll be prepared to ace your JavaScript job interview and even impress your interviewers with some bits that, maybe, they didn't know.

ECMAScript 5

Revision 3 of ECMAScript is the one you can take for granted to be implemented in all browsers and environments. Revision 4 was skipped and revision 5 (let's call it ES5 for short) was officially accepted in December 2009.

ES5 introduces some new objects and properties and also the so-called "strict mode". Strict mode is a subset of the language that excludes deprecated features. The strict mode is opt-in and not required, meaning that if you want your code to run in the strict mode, you declare your intention using (once per function, or once for the whole program) the following string:

```
"use strict";
```

This is just a JavaScript string, and it's OK to have strings floating around unassigned to any variable. As a result, older browsers that don't "speak" ES5 will simply ignore it, so this strict mode is backwards compatible and won't break older browsers.

In future versions, strict mode is likely to become the default or the only mode. For the time being, it's optional.

For backwards compatibility, all the examples in this book work in ES3, but at the same time, all the code in the book is written so that it will run without warnings in ES5's strict mode. Additionally, any ES5-specific parts will be clearly marked. *Chapter 11*, *Built-in Objects*, lists the new additions to ES5 in detail.

Object-oriented programming

Here's a quick table summarizing the object oriented programming concepts discussed in the previous module:

Feature	**Illustrates concept**
Bob is a man (an object).	Objects
Bob's date of birth is June 1, 1980, gender: male, and hair: black.	Properties
Bob can eat, sleep, drink, dream, talk, and calculate his own age.	Methods
Bob is an instance of the `Programmer` class.	Class (in classical OOP)
Bob is based on another object, called `Programmer`.	Prototype (in prototypal OOP)
Bob holds data (such as `birth_date`) and methods that work with the data (such as `calculateAge()`).	Encapsulation

Feature	Illustrates concept
You don't need to know how the calculation method works internally. The object might have some private data, such as the number of days in February in a leap year. You don't know, nor do you want to know.	Information hiding
Bob is part of a `WebDevTeam` object, together with Jill, a `Designer` object, and Jack, a `ProjectManager` object.	Aggregation and composition
`Designer`, `ProjectManager`, and `Programmer` are all based on and extend a `Person` object.	Inheritance
You can call the methods `Bob.talk()`, `Jill.talk()`, and `Jack.talk()` and they'll all work fine, albeit producing different results (Bob will probably talk more about performance, Jill about beauty, and Jack about deadlines). Each object inherited the method `talk` from `Person` and customized it.	Polymorphism and method overriding

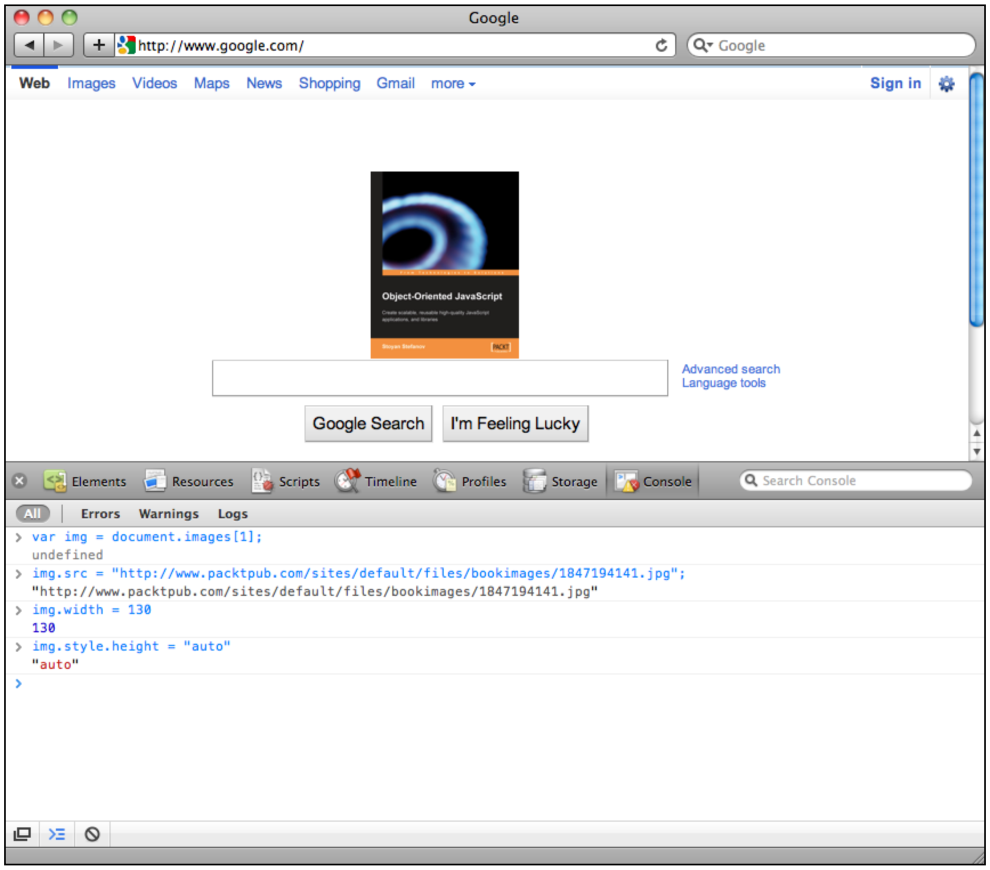
Setting up your training environment

This book takes a "do-it-yourself" approach when it comes to writing code, because I firmly believe that the best way to really learn a programming language is by writing code. There are no cut-and-paste-ready code downloads that you simply put in your pages. On the contrary, you're expected to type in code, see how it works, and then tweak it and play around with it. When trying out the code examples, you're encouraged to enter the code into a JavaScript console. Let's see how you go about doing this.

As a developer, you most likely already have a number of web browsers installed on your system such as Firefox, Safari, Chrome, or Internet Explorer. All modern browsers have a JavaScript console feature, which you'll use throughout the book to help you learn and experiment with the language. More specifically, this book uses WebKit's console (available in Safari and Chrome), but the examples should work in any other console.

WebKit's Web Inspector

This example shows how you can use the console to type in some code that swaps the logo on the `google.com` home page with an image of your choice. As you can see, you can test your JavaScript code live on any page.



```
> var img = document.images[1];
  undefined
> img.src = "http://www.packtpub.com/sites/default/files/bookimages/1847194141.jpg";
  "http://www.packtpub.com/sites/default/files/bookimages/1847194141.jpg"
> img.width = 130
  130
> img.style.height = "auto"
  "auto"
>
```

In order to bring up the console in Chrome or Safari, right-click anywhere on a page and select **Inspect Element**. The additional window that shows up is the Web Inspector feature. Select the **Console** tab and you're ready to go.

You type code directly into the console, and when you press *Enter*, your code is executed. The return value of the code is printed in the console. The code is executed in the context of the currently loaded page, so for example, if you type `location.href`, it will return the URL of the current page.

The console also has an autocomplete feature. It works similar to the normal command line prompt in your operating system. If, for example, you type `docu` and hit the *Tab* key or the right arrow key, `docu` will be autocompleted to `document`. Then, if you type `.` (the dot operator), you can iterate through all the available properties and methods you can call on the `document` object.

By using the up and down arrow keys, you can go through the list of already executed commands and bring them back in the console.

The console gives you only one line to type in, but you can execute several JavaScript statements by separating them with semicolons. If you need more lines, you can press *Shift* + *Enter* to go to a new line without executing the result just yet.

JavaScriptCore on a Mac

On a Mac, you don't actually need a browser; you can explore JavaScript directly from your command line **Terminal** application.

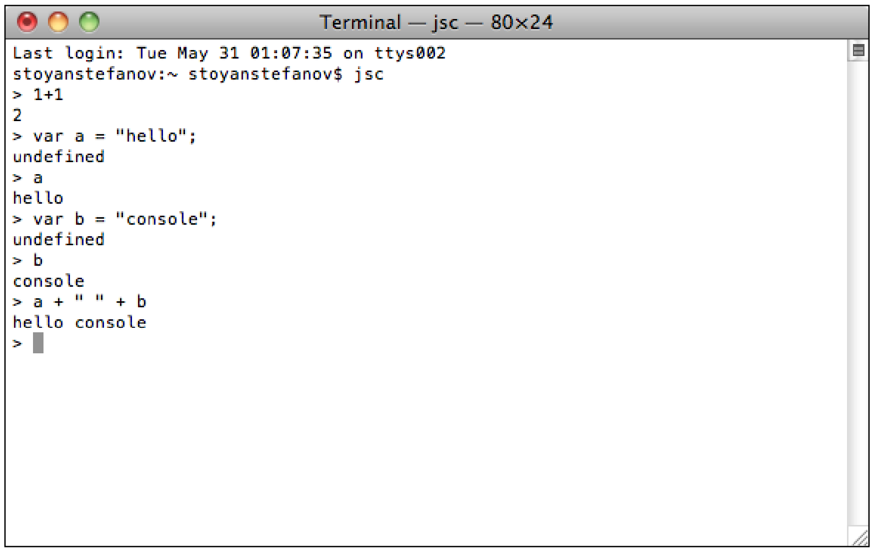
If you've never used **Terminal**, you can simply search for it in the Spotlight search. Once you've launched it, type:

```
alias jsc='/System/Library/Frameworks/JavaScriptCore.framework/Versions/
Current/Resources/jsc'
```

This command makes an alias to the little **jsc** application, which stands for "JavaScriptCore" and is part of the WebKit engine. JavaScriptCore is shipped together with Mac operating systems.

You can add the `alias` line shown previously to your `~/.profile` file so that `jsc` is always there when you need it.

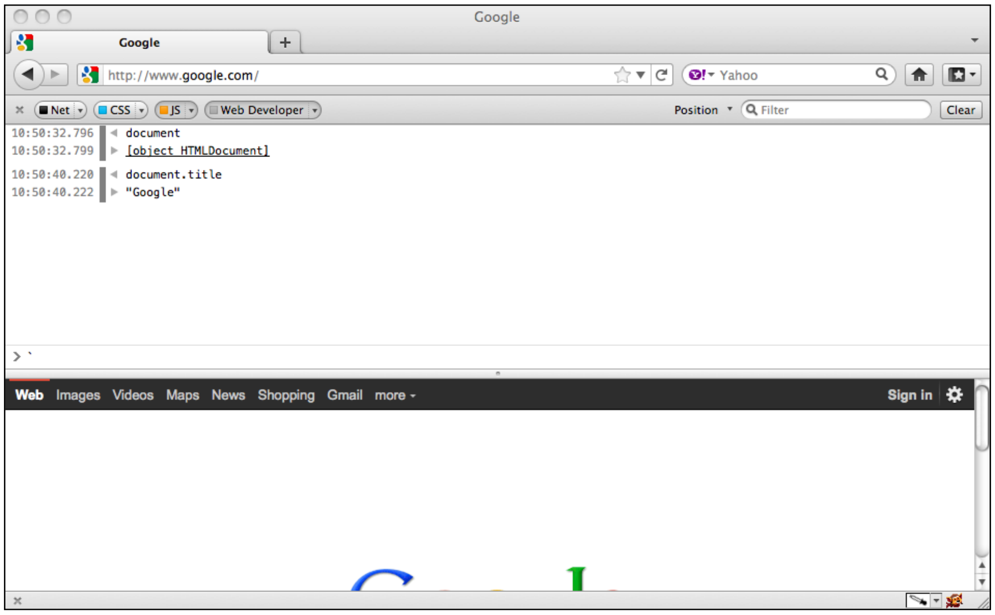
Now, in order to start the interactive shell, you simply type `jsc` from any directory. Then you can type JavaScript expressions, and when you hit *Enter*, you'll see the result of the expression.



```
Last login: Tue May 31 01:07:35 on ttys002
stoyanstefanov:~ stoyanstefanov$ jsc
> 1+1
2
> var a = "hello";
undefined
> a
hello
> var b = "console";
undefined
> b
console
> a + " " + b
hello console
> 
```

More consoles

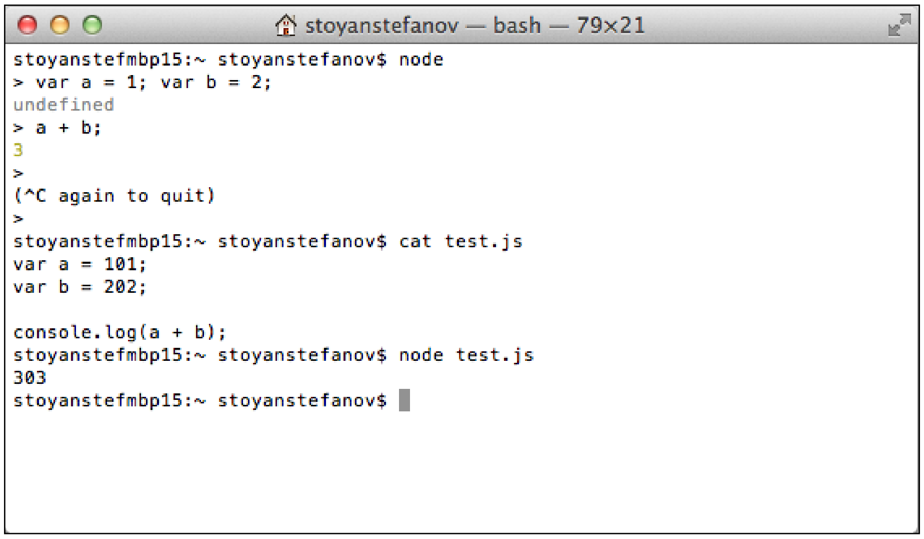
All modern browsers have consoles built in. You have seen the Chrome/Safari console previously. In any Firefox version, you can install the Firebug extension, which comes with a console. Additionally, in newer Firefox releases, there's a console built in and accessible via the **Tools/Web Developer/Web Console** menu.



```
10:50:32.796  document
10:50:32.799  [object HTMLDocument]
10:50:40.220  document.title
10:50:40.222  "Google"
```

Internet Explorer, since Version 8, has an F12 Developer Tools feature, which has a console in its **Script** tab.

It's also a good idea to familiarize yourself with `Node.js`, and you can start by trying out its console. Install `Node.js` from `http://nodejs.org` and try the console in your command prompt (terminal).



```
stoyanstefmbp15:~ stoyanstefanov$ node
> var a = 1; var b = 2;
undefined
> a + b;
3
>
(^C again to quit)
>
stoyanstefmbp15:~ stoyanstefanov$ cat test.js
var a = 101;
var b = 202;

console.log(a + b);
stoyanstefmbp15:~ stoyanstefanov$ node test.js
303
stoyanstefmbp15:~ stoyanstefanov$
```

As you can see, you can use the `Node.js` console to try out quick examples. But, you can also write longer shell scripts (`test.js` in the screenshot) and run them with the `scriptname.js` node.

Summary

In this chapter, you learned about how JavaScript came to be and where it is today. You were also introduced to object-oriented programming concepts and have seen how JavaScript is not a class-based OO language, but a prototype-based one. Finally, you learned how to use your training environment—the JavaScript console. Now you're ready to dive into JavaScript and learn how to use its powerful OO features. But let's start from the beginning.

The next chapter will guide you through the data types in JavaScript (there are just a few), conditions, loops, and arrays. If you think you know these topics, feel free to skip the next chapter, but not before you make sure you can complete the few short exercises at the end of the chapter.

2 Primitive Data Types, Arrays, Loops, and Conditions

Before diving into the object-oriented features of JavaScript, let's first take a look at some of the basics. This chapter walks you through the following:

- The primitive data types in JavaScript, such as strings and numbers
- Arrays
- Common operators, such as `+`, `-`, `delete`, and `typeof`
- Flow control statements, such as loops and if-else conditions

Variables

Variables are used to store data; they are placeholders for concrete values. When writing programs, it's convenient to use variables instead of the actual data, as it's much easier to write `pi` instead of `3.141592653589793`, especially when it happens several times inside your program. The data stored in a variable can be changed after it was initially assigned, hence the name "variable". You can also use variables to store data that is unknown to you while you write the code, such as the result of a later operation.

Using a variable requires two steps. You need to:

- Declare the variable
- Initialize it, that is, give it a value

To declare a variable, you use the `var` statement, like this:

```
var a;
var thisIsAVariable;
var _and_this_too;
var mix12three;
```

For the names of the variables, you can use any combination of letters, numbers, the underscore character, and the dollar sign. However, you can't start with a number, which means that this is invalid:

```
var 2three4five;
```

To initialize a variable means to give it a value for the first (initial) time. You have two ways to do so:

- Declare the variable first, then initialize it
- Declare and initialize it with a single statement

An example of the latter is:

```
var a = 1;
```

Now the variable named `a` contains the value `1`.

You can declare (and optionally initialize) several variables with a single `var` statement; just separate the declarations with a comma:

```
var v1, v2, v3 = 'hello', v4 = 4, v5;
```

For readability, this is often written using one variable per line:

```
var v1,
    v2,
    v3 = 'hello',
    v4 = 4,
    v5;
```

The $ character in variable names

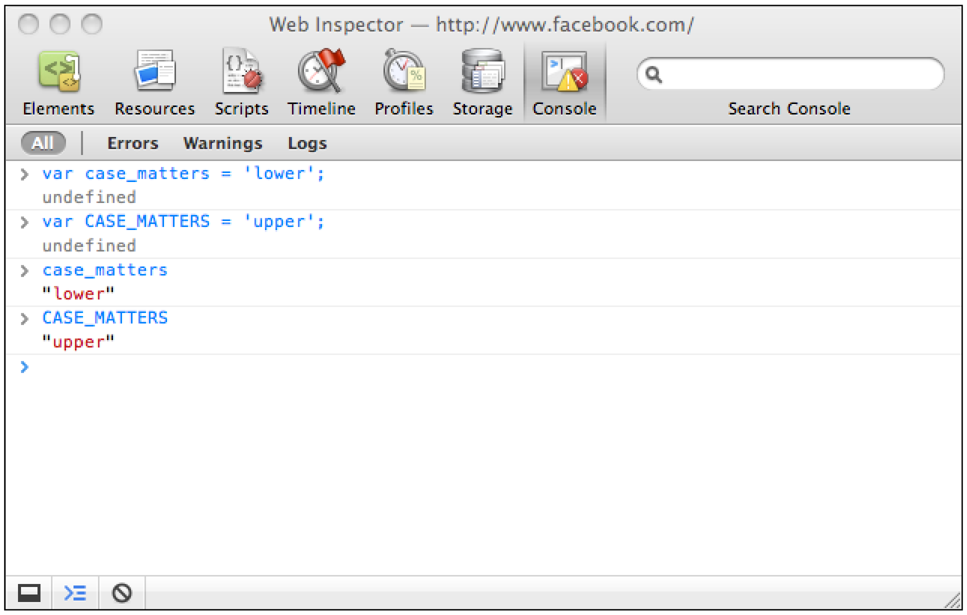
You may see the dollar sign character ($) used in variable names, as in `$myvar` or less commonly `my$var`. This character is allowed to appear anywhere in a variable name, although previous versions of the ECMA standard discouraged its use in handwritten programs and suggested it should only be used in generated code (programs written by other programs). This suggestion is not well respected by the JavaScript community, and $ is in fact commonly used in practice as a function name.

Variables are case sensitive

Variable names are case sensitive. You can easily verify this statement using your JavaScript console. Try typing this, pressing *Enter* after each line:

```
var case_matters = 'lower';
var CASE_MATTERS = 'upper';
case_matters;
CASE_MATTER;
```

To save keystrokes, when you enter the third line, you can type `case` and press the *Tab* key (or right-arrow key). The console autocompletes the variable name to **case_matters**. Similarly, for the last line, type `CASE` and press *Tab*. The end result is shown in the following figure:



```
> var case_matters = 'lower';
  undefined
> var CASE_MATTERS = 'upper';
  undefined
> case_matters
  "lower"
> CASE_MATTERS
  "upper"
>
```

Throughout the rest of this book, only the code for the examples is given instead of a screenshot, like so:

```
> var case_matters = 'lower';
> var CASE_MATTERS = 'upper';
> case_matters;
"lower"

> CASE_MATTERS;
"upper"
```

The greater-than signs (`>`) show the code that you type; the rest is the result as printed in the console. Again, remember that when you see such code examples, you're strongly encouraged to type in the code yourself. Then, you can experiment by tweaking it a little here and there to get a better feeling of how exactly it works.

You can see in the screenshot that sometimes what you type in the console results in the word **undefined**. You can simply ignore this, but if you're curious, here's what happens: when evaluating (executing) what you type, the console prints the returned value. Some expressions (such as `var a = 1;`) don't return anything explicitly, in which case they implicitly return the special value **undefined** (more on it in a bit). When an expression returns some value (for example `case_matters` in the previous example or something such as `1 + 1`), the resulting value is printed out. Not all consoles print the **undefined** value, for example the Firebug console.

Operators

Operators take one or two values (or variables), perform an operation, and return a value. Let's check out a simple example of using an operator, just to clarify the terminology:

```
> 1 + 2;
3
```

In this code:

- `+` is the operator
- The operation is addition
- The input values are `1` and `2` (the input values are also called operands)
- The result value is `3`
- The whole thing is called an expression

Instead of using the values `1` and `2` directly in the expression, you can use variables. You can also use a variable to store the result of the operation, as the following example demonstrates:

```
> var a = 1;
> var b = 2;
> a + 1;
2
```

```
> b + 2;
4

> a + b;
3

> var c = a + b;
> c;
3
```

The following table lists the basic arithmetic operators:

Operator symbol	Operation	Example
`+`	Addition	`> 1 + 2;` **3**
`-`	Subtraction	`> 99.99 - 11;` **88.99**
`*`	Multiplication	`> 2 * 3;` **6**
`/`	Division	`> 6 / 4;` **1.5**
`%`	Modulo, the remainder of a division	`> 6 % 3;` **0** `> 5 % 3;` **2** It's sometimes useful to test if a number is even or odd. Using the modulo operator, it's easy to do just that. All odd numbers return 1 when divided by 2, while all even numbers return 0. `> 4 % 2;` **0** `> 5 % 2;` **1**

Operator symbol	Operation	Example

`++`	Increment a value by 1	Post-increment is when the input value is incremented after it's returned.
		`> var a = 123;`
		`> var b = a++;`
		`> b;`
		123
		`> a;`
		124
		The opposite is pre-increment. The input value is incremented by 1 first and then returned.
		`> var a = 123;`
		`> var b = ++a;`
		`> b;`
		124
		`> a;`
		124

`--`	Decrement a value by 1	Post-decrement:
		`> var a = 123;`
		`> var b = a--;`
		`> b;`
		123
		`> a;`
		122
		Pre-decrement:
		`> var a = 123;`
		`> var b = --a;`
		`> b;`
		122
		`> a;`
		122

`var a = 1;` is also an operation; it's the simple assignment operation, and `=` is the **simple assignment operator**.

There is also a family of operators that are a combination of an assignment and an arithmetic operator. These are called **compound operators**. They can make your code more compact. Let's see some of them with examples:

```
> var a = 5;
> a += 3;
8
```

In this example, `a += 3;` is just a shorter way of doing `a = a + 3;`:

```
> a -= 3;
5
```

Here, `a -= 3;` is the same as `a = a - 3;`.

Similarly:

```
> a *= 2;
10

> a /= 5;
2

> a %= 2;
0
```

In addition to the arithmetic and assignment operators discussed previously, there are other types of operators, as you'll see later in this and the following chapters.

Best practice

Always end your expressions with a semicolon. JavaScript has a semicolon insertion mechanism where it can add the semicolon if you forget it at the end of a line. However, this can also be a source of errors, so it's best to make sure you always explicitly state where you want to terminate your expressions. In other words, both expressions, `> 1 + 1` and `> 1 + 1;`, will work; but, throughout the book you'll always see the second type, terminated with a semicolon, just to emphasize this habit.

Primitive data types

Any value that you use is of a certain type. In JavaScript, there are just a few primitive data types:

1. Number: This includes floating point numbers as well as integers. For example, these values are all numbers: `1`, `100`, `3.14`.
2. String: These consist of any number of characters, for example `"a"`, `"one"`, and `"one 2 three"`.
3. Boolean: This can be either `true` or `false`.
4. Undefined: When you try to access a variable that doesn't exist, you get the special value `undefined`. The same happens when you declare a variable without assigning a value to it yet. JavaScript initializes the variable behind the scenes with the value `undefined`. The undefined data type can only have one value – the special value `undefined`.
5. Null: This is another special data type that can have only one value, namely the `null` value. It means no value, an empty value, or nothing. The difference with `undefined` is that if a variable has a value `null`, it's still defined, it just so happens that its value is nothing. You'll see some examples shortly.

Any value that doesn't belong to one of the five primitive types listed here is an object. Even `null` is considered an object, which is a little awkward—having an object (something) that is actually nothing. We'll learn more on objects in *Chapter 4, Objects*, but for the time being, just remember that in JavaScript the data types are either:

- Primitive (the five types listed previously)
- Non-primitive (objects)

Finding out the value type – the typeof operator

If you want to know the type of a variable or a value, you use the special `typeof` operator. This operator returns a string that represents the data type. The return values of using `typeof` are one of the following:

- `"number"`
- `"string"`
- `"boolean"`

- `"undefined"`
- `"object"`
- `"function"`

In the next few sections, you'll see `typeof` in action using examples of each of the five primitive data types.

Numbers

The simplest number is an integer. If you assign `1` to a variable and then use the `typeof` operator, it returns the string `"number"`:

```
> var n = 1;
> typeof n;
"number"

> n = 1234;
> typeof n;
"number"
```

In the preceding example, you can see that the second time you set a variable's value, you don't need the `var` statement.

Numbers can also be floating point (decimals):

```
> var n2 = 1.23;
> typeof n;
"number"
```

You can call `typeof` directly on the value without assigning it to a variable first:

```
> typeof 123;
"number"
```

Octal and hexadecimal numbers

When a number starts with a 0, it's considered an octal number. For example, the octal `0377` is the decimal `255`:

```
> var n3 = 0377;
> typeof n3;
"number"

> n3;
255
```

The last line in the preceding example prints the decimal representation of the octal value.

While you may not be intimately familiar with octal numbers, you've probably used hexadecimal values to define colors in CSS stylesheets.

In CSS, you have several options to define a color, two of them being:

- Using decimal values to specify the amount of R (red), G (green), and B (blue) ranging from 0 to 255. For example, `rgb(0, 0, 0)` is black and `rgb(255, 0, 0)` is red (maximum amount of red and no green or blue).
- Using hexadecimals and specifying two characters for each R, G, and B value. For example, `#000000` is black and `#ff0000` is red. This is because `ff` is the hexadecimal value for 255.

In JavaScript, you put `0x` before a hexadecimal value (also called hex for short):

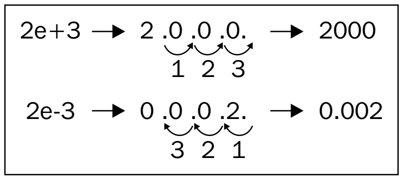
```
> var n4 = 0x00;
> typeof n4;
"number"

> n4;
0

> var n5 = 0xff;
> typeof n5;
"number"

> n5;
255
```

Exponent literals

`1e1` (also written as `1e+1` or `1E1` or `1E+1`) represents the number one with one zero after it, or in other words, 10. Similarly, `2e+3` means the number 2 with 3 zeros after it, or 2000:

```
> 1e1;
10

> 1e+1;
10

> 2e+3;
2000

> typeof 2e+3;
"number"
```

`2e+3` means moving the decimal point three digits to the right of the number 2. There's also `2e-3`, meaning you move the decimal point three digits to the left of the number 2:



```
> 2e-3;
0.002

> 123.456E-3;
0.123456

> typeof 2e-3;
"number"
```

Infinity

There is a special value in JavaScript called `Infinity`. It represents a number too big for JavaScript to handle. `Infinity` is indeed a number, as typing `typeof Infinity` in the console will confirm. You can also quickly check that a number with 308 zeros is ok, but 309 zeros is too much. To be precise, the biggest number JavaScript can handle is `1.7976931348623157e+308`, while the smallest is `5e-324`.

```
> Infinity;
Infinity

> typeof Infinity;
"number"

> 1e309;
Infinity

> 1e308;
1e+308
```

Dividing by zero gives you infinity:

```
> var a = 6 / 0;
> a;
Infinity
```

Infinity is the biggest number (or rather a little bigger than the biggest), but how about the smallest? It's infinity with a minus sign in front of it; minus infinity:

```
> var i = -Infinity;
> i;
-Infinity

> typeof i;
"number"
```

Does this mean you can have something that's exactly twice as big as `Infinity`, from 0 up to infinity and then from 0 down to minus infinity? Well, not really. When you sum infinity and minus infinity, you don't get 0, but something that is called `NaN` (Not a Number):

```
> Infinity - Infinity;
NaN

> -Infinity + Infinity;
NaN
```

Any other arithmetic operation with `Infinity` as one of the operands gives you `Infinity`:

```
> Infinity - 20;
Infinity

> -Infinity * 3;
-Infinity

> Infinity / 2;
Infinity

> Infinity - 99999999999999999;
Infinity
```

NaN

What was this `NaN` in the previous example? It turns out that despite its name, "Not a Number", `NaN` is a special value that is also a number:

```
> typeof NaN;
"number"

> var a = NaN;
> a;
NaN
```

You get `NaN` when you try to perform an operation that assumes numbers, but the operation fails. For example, if you try to multiply `10` by the character `"f"`, the result is `NaN`, because `"f"` is obviously not a valid operand for a multiplication:

```
> var a = 10 * "f";
> a;
NaN
```

`NaN` is contagious, so if you have even one `NaN` in your arithmetic operation, the whole result goes down the drain:

```
> 1 + 2 + NaN;
NaN
```

Strings

A string is a sequence of characters used to represent text. In JavaScript, any value placed between single or double quotes is considered a string. This means that `1` is a number, but `"1"` is a string. When used with strings, `typeof` returns the string `"string"`:

```
> var s = "some characters";
> typeof s;
"string"
```

```
> var s = 'some characters and numbers 123 5.87';
> typeof s;
"string"
```

Here's an example of a number used in the string context:

```
> var s = '1';
> typeof s;
"string"
```

If you put nothing in quotes, it's still a string (an empty string):

```
> var s = ""; typeof s;
"string"
```

As you already know, when you use the plus sign with two numbers, this is the arithmetic addition operation. However, if you use the plus sign with strings, this is a string concatenation operation, and it returns the two strings glued together:

```
> var s1 = "web";
> var s2 = "site";
> var s = s1 + s2;
> s;
"website"

> typeof s;
"string"
```

The dual purpose of the + operator is a source of errors. Therefore, if you intend to concatenate strings, it's always best to make sure that all of the operands are strings. The same applies for addition; if you intend to add numbers, make sure the operands are numbers. You'll learn various ways to do so further in the chapter and the book.

String conversions

When you use a number-like string (for example `"1"`) as an operand in an arithmetic operation, the string is converted to a number behind the scenes. This works for all arithmetic operations except addition, because of its ambiguity:

```
> var s = '1';
> s = 3 * s;
> typeof s;
"number"

> s;
3

> var s = '1';
> s++;
> typeof s;
"number"

> s;
2
```

A lazy way to convert any number-like string to a number is to multiply it by 1 (another way is to use a function called `parseInt()`, as you'll see in the next chapter):

```
> var s = "100"; typeof s;
"string"

> s = s * 1;
100

> typeof s;
"number"
```

If the conversion fails, you'll get `NaN`:

```
> var movie = '101 dalmatians';
> movie * 1;
NaN
```

You convert a string to a number by multiplying by 1. The opposite—converting anything to a string—can be done by concatenating it with an empty string:

```
> var n = 1;
> typeof n;
"number"

> n = "" + n;
"1"

> typeof n;
"string"
```

Special strings

There are also strings with special meanings, as listed in the following table:

String	Meaning	Example
`\\` `\'` `\"`	\ is the escape character. When you want to have quotes inside your string, you escape them so that JavaScript doesn't think they mean the end of the string. If you want to have an actual backslash in the string, escape it with another backslash.	`> var s = 'I don't know';` This is an error, because JavaScript thinks the string is `I don` and the rest is invalid code. The following are valid: • `> var s = 'I don\'t know';` • `> var s = "I don\'t know";` • `> var s = "I don't know";` • `> var s = '"Hello", he said.';` • `> var s = "\"Hello\", he said.";` Escaping the escape: `> var s = "1\\2"; s;` **"1\2"**

String	Meaning	Example

`\n`	End of line.	`> var s = '\n1\n2\n3\n';`
		`> s;`
		"
		1
		2
		3
		"

`\r`	Carriage return.	Consider the following statements: • > var s = '1\r2'; • > var s = '1\n\r2'; • > var s = '1\r\n2'; The result of all of these is: `> s;` **"1** **2"**

`\t`	Tab.	`> var s = "1\t2";`
		`> s;`
		"1 2"

`\u`	`\u` followed by a character code allows you to use Unicode.	Here's my name in Bulgarian written with Cyrillic characters:
		`> "\u0421\u0442\u043E\`
		`u044F\u043D";`
		"Стоян"

There are also additional characters that are rarely used: `\b` (backspace), `\v` (vertical tab), and `\f` (form feed).

Booleans

There are only two values that belong to the Boolean data type: the values `true` and `false`, used without quotes:

```
> var b = true;
> typeof b;
"boolean"

> var b = false;
> typeof b;
"boolean"
```

If you quote `true` or `false`, they become strings:

```
> var b = "true";
> typeof b;
"string"
```

Logical operators

There are three operators, called logical operators, that work with Boolean values. These are:

- `!` – logical NOT (negation)
- `&&` – logical AND
- `||` – logical OR

You know that when something is not true, it must be false. Here's how this is expressed using JavaScript and the logical `!` operator:

```
> var b = !true;
> b;
false
```

If you use the logical NOT twice, you get the original value:

```
> var b = !!true;
> b;
true
```

If you use a logical operator on a non-Boolean value, the value is converted to Boolean behind the scenes:

```
> var b = "one";
> !b;
false
```

In the preceding case, the string value `"one"` is converted to a Boolean, `true`, and then negated. The result of negating `true` is `false`. In the next example, there's a double negation, so the result is `true`:

```
> var b = "one";
> !!b;
true
```

You can convert any value to its Boolean equivalent using a double negation. Understanding how any value converts to a Boolean is important. Most values convert to `true` with the exception of the following, which convert to `false`:

- The empty string `""`
- `null`
- `undefined`
- The number `0`
- The number `NaN`
- The Boolean `false`

These six values are referred to as falsy, while all others are truthy (including, for example, the strings `"0"`, `" "`, and `"false"`).

Let's see some examples of the other two operators—the logical AND (`&&`) and the logical OR (`||`). When you use `&&`, the result is `true` only if all of the operands are `true`. When you use `||`, the result is `true` if at least one of the operands is `true`:

```
> var b1 = true, b2 = false;
> b1 || b2;
true

> b1 && b2;
false
```

Here's a list of the possible operations and their results:

Operation	**Result**
`true && true`	`true`
`true && false`	`false`
`false && true`	`false`
`false && false`	`false`
`true || true`	`true`
`true || false`	`true`
`false || true`	`true`
`false || false`	`false`

You can use several logical operations one after the other:

```
> true && true && false && true;
false
```

```
> false || true || false;
true
```

You can also mix `&&` and `||` in the same expression. In such cases, you should use parentheses to clarify how you intend the operation to work. Consider these:

```
> false && false || true && true;
true

> false && (false || true) && true;
false
```

Operator precedence

You might wonder why the previous expression (`false && false || true && true`) returned `true`. The answer lies in the operator precedence. As you know from mathematics:

```
> 1 + 2 * 3;
7
```

This is because multiplication has higher precedence over addition, so `2 * 3` is evaluated first, as if you typed:

```
> 1 + (2 * 3);
7
```

Similarly for logical operations, `!` has the highest precedence and is executed first, assuming there are no parentheses that demand otherwise. Then, in the order of precedence, comes `&&` and finally `||`. In other words, the following two code snippets are the same:

```
> false && false || true && true;
true
```

and

```
> (false && false) || (true && true);
true
```

Best practice

Use parentheses instead of relying on operator precedence. This makes your code easier to read and understand.

The ECMAScript standard defines the precedence of operators. While it may be a good memorization exercise, this book doesn't offer it. First of all, you'll forget it, and second, even if you manage to remember it, you shouldn't rely on it. The person reading and maintaining your code will likely be confused.

Lazy evaluation

If you have several logical operations one after the other, but the result becomes clear at some point before the end, the final operations will not be performed because they don't affect the end result. Consider this:

```
> true || false || true || false || true;
true
```

Since these are all OR operations and have the same precedence, the result will be `true` if at least one of the operands is `true`. After the first operand is evaluated, it becomes clear that the result will be `true`, no matter what values follow. So, the JavaScript engine decides to be lazy (OK, efficient) and avoids unnecessary work by evaluating code that doesn't affect the end result. You can verify this short-circuiting behavior by experimenting in the console:

```
> var b = 5;
> true || (b = 6);
true

> b;
5

> true && (b = 6);
6

> b;
6
```

This example also shows another interesting behavior: if JavaScript encounters a non-Boolean expression as an operand in a logical operation, the non-Boolean is returned as a result:

```
> true || "something";
true

> true && "something";
"something"

> true && "something" && true;
true
```

This behavior is not something you should rely on because it makes the code harder to understand. It's common to use this behavior to define variables when you're not sure whether they were previously defined. In the next example, if the variable `mynumber` is defined, its value is kept; otherwise, it's initialized with the value `10`:

```
> var mynumber = mynumber || 10;
> mynumber;
10
```

This is simple and looks elegant, but be aware that it's not completely foolproof. If `mynumber` is defined and initialized to 0 (or to any of the six falsy values), this code might not behave as you expect:

```
> var mynumber = 0;
> var mynumber = mynumber || 10;
> mynumber;
10
```

Comparison

There's another set of operators that all return a Boolean value as a result of the operation. These are the comparison operators. The following table lists them together with example uses:

Operator symbol	Description	Example
`==`	Equality comparison: Returns true when both operands are equal. The operands are converted to the same type before being compared. Also called loose comparison.	`> 1 == 1;` **true** `> 1 == 2;` **false** `> 1 == '1';` **true**
`===`	Equality and type comparison: Returns `true` if both operands are equal and of the same type. It's better and safer to compare this way because there's no behind-the-scenes type conversions. It is also called strict comparison.	`> 1 === '1';` **false** `> 1 === 1;` **true**

Operator symbol	Description	Example
`!=`	Non-equality comparison: Returns `true` if the operands are not equal to each other (after a type conversion).	`> 1 != 1;` **false** `> 1 != '1';` **false** `> 1 != '2';` **true**
`!==`	Non-equality comparison without type conversion: Returns `true` if the operands are not equal or if they are of different types.	`> 1 !== 1;` **false** `> 1 !== '1';` **true**
`>`	Returns `true` if the left operand is greater than the right one.	`> 1 > 1;` **false** `> 33 > 22;` **true**
`>=`	Returns `true` if the left operand is greater than or equal to the right one.	> 1 >= 1; **true**
`<`	Returns `true` if the left operand is less than the right one.	> 1 < 1; **false** `> 1 < 2;` **true**
`<=`	Returns `true` if the left operand is less than or equal to the right one.	> 1 <= 1; **true** `> 1 <= 2;` **true**

Note that `NaN` is not equal to anything, not even itself:

```
> NaN == NaN;
false
```

Undefined and null

If you try to use a non-existing variable, you'll get an error:

```
> foo;
ReferenceError: foo is not defined
```

Using the `typeof` operator on a non-existing variable is not an error. You get the string `"undefined"` back:

```
> typeof foo;
"undefined"
```

If you declare a variable without giving it a value, this is, of course, not an error. But, the `typeof` still returns `"undefined"`:

```
> var somevar;
> somevar;
> typeof somevar;
"undefined"
```

This is because when you declare a variable without initializing it, JavaScript automatically initializes it with the value `undefined`:

```
> var somevar;
> somevar === undefined;
true
```

The `null` value, on the other hand, is not assigned by JavaScript behind the scenes; it's assigned by your code:

```
> var somevar = null;
null

> somevar;
null

> typeof somevar;
"object"
```

Although the difference between `null` and `undefined` is small, it could be critical at times. For example, if you attempt an arithmetic operation, you get different results:

```
> var i = 1 + undefined;
> i;
NaN

> var i = 1 + null;
> i;
1
```

This is because of the different ways `null` and `undefined` are converted to the other primitive types. The following examples show the possible conversions:

- Conversion to a number:

  ```
  > 1 * undefined;
  NaN

  > 1 * null;
  0
  ```

- Conversion to a Boolean:

  ```
  > !!undefined;
  false

  > !!null;
  false
  ```

- Conversion to a string:

  ```
  > "value: " + null;
  "value: null"

  > "value: " + undefined;
  "value: undefined"
  ```

Primitive data types recap

Let's quickly summarize some of the main points discussed so far:

- There are five primitive data types in JavaScript:
  - Number
  - String
  - Boolean
  - Undefined
  - Null
- Everything that is not a primitive data type is an object
- The primitive number data type can store positive and negative integers or floats, hexadecimal numbers, octal numbers, exponents, and the special numbers `NaN`, `Infinity`, and `-Infinity`
- The string data type contains characters in quotes
- The only values of the Boolean data type are `true` and `false`
- The only value of the null data type is the value `null`

- The only value of the undefined data type is the value `undefined`
- All values become `true` when converted to a Boolean, with the exception of the six falsy values:
    - `""`
    - `null`
    - `undefined`
    - `0`
    - `NaN`
    - `false`

Arrays

Now that you know about the basic primitive data types in JavaScript, it's time to move to a more powerful data structure—the array.

So, what is an array? It's simply a list (a sequence) of values. Instead of using one variable to store one value, you can use one array variable to store any number of values as elements of the array.

To declare a variable that contains an empty array, you use square brackets with nothing between them:

```
> var a = [];
```

To define an array that has three elements, you do this:

```
> var a = [1, 2, 3];
```

When you simply type the name of the array in the console, you get the contents of your array:

```
> a;
[1, 2, 3]
```

Now the question is how to access the values stored in these array elements. The elements contained in an array are indexed with consecutive numbers starting from zero. The first element has index (or position) 0, the second has index 1, and so on. Here's the three-element array from the previous example:

Index	Value
0	1
1	2
2	3

To access an array element, you specify the index of that element inside square brackets. So, `a[0]` gives you the first element of the array `a`, `a[1]` gives you the second, and so on:

```
> a[0];
1

> a[1];
2
```

Adding/updating array elements

Using the index, you can also update the values of the elements of the array. The next example updates the third element (index 2) and prints the contents of the new array:

```
> a[2] = 'three';
"three"

> a;
[1, 2, "three"]
```

You can add more elements by addressing an index that didn't exist before:

```
> a[3] = 'four';
"four"

> a;
[1, 2, "three", "four"]
```

If you add a new element, but leave a gap in the array, those elements in between don't exist and return the `undefined` value if accessed. Check out this example:

```
> var a = [1, 2, 3];
> a[6] = 'new';
"new"

> a;
[1, 2, 3, undefined x 3, "new"]
```

Deleting elements

To delete an element, you use the `delete` operator. However, after the deletion, the length of the array does not change. In a sense, you get a hole in the array:

```
> var a = [1, 2, 3];
> delete a[1];
true

> a;
[1, undefined, 3]

> typeof a[1];
"undefined"
```

Arrays of arrays

Arrays can contain all types of values, including other arrays:

```
> var a = [1, "two", false, null, undefined];
> a;
[1, "two", false, null, undefined]

> a[5] = [1, 2, 3];
[1, 2, 3]

> a;
[1, "two", false, null, undefined, Array[3]]
```

The **Array[3]** in the result is clickable in the console and it expands the array values. Let's see an example where you have an array of two elements, both of them being other arrays:

```
> var a = [[1, 2, 3], [4, 5, 6]];
> a;
[Array[3], Array[3]]
```

The first element of the array is `a[0]`, and it's also an array:

```
> a[0];
[1, 2, 3]
```

To access an element in the nested array, you refer to the element index in another set of square brackets:

```
> a[0][0];
1

> a[1][2];
6
```

Note that you can use the array notation to access individual characters inside a string:

```
> var s = 'one';
> s[0];
"o"

> s[1];
"n"

> s[2];
"e"
```

> Array access to strings has been supported by many browsers for a while (not older IEs), but has been officially recognized only as late as ECMAScript 5.

There are more ways to have fun with arrays (and you'll get to those in *Chapter 4, Objects*), but let's stop here for now, remembering that:

- An array is a data store
- An array contains indexed elements
- Indexes start from zero and increment by one for each element in the array
- To access an element of an array, you use its index in square brackets
- An array can contain any type of data, including other arrays

Conditions and loops

Conditions provide a simple but powerful way to control the flow of code execution. **Loops** allow you to perform repetitive operations with less code. Let's take a look at:

- `if` conditions
- `switch` statements
- `while`, `do-while`, `for`, and `for-in` loops

The examples in the following sections require you to switch to the multiline Firebug console. Or, if you use the WebKit console, use *Shift* + *Enter* instead of *Enter* to add a new line.

The if condition

Here's a simple example of an `if` condition:

```
var result = '', a = 3;
if (a > 2) {
  result = 'a is greater than 2';
}
```

The parts of the `if` condition are:

- The `if` statement
- A condition in parentheses—"is `a` greater than 2?"
- A block of code wrapped in `{}` that executes if the condition is satisfied

The condition (the part in parentheses) always returns a Boolean value, and may also contain the following:

- A logical operation: `!`, `&&`, or `||`
- A comparison, such as `===`, `!=`, `>`, and so on
- Any value or variable that can be converted to a Boolean
- A combination of the above

The else clause

There can also be an optional `else` part of the `if` condition. The `else` statement is followed by a block of code that runs if the condition evaluates to `false`:

```
if (a > 2) {
  result = 'a is greater than 2';
} else {
  result = 'a is NOT greater than 2';
}
```

In between the `if` and the `else`, there can also be an unlimited number of `else if` conditions. Here's an example:

```
if (a > 2 || a < -2) {
  result = 'a is not between -2 and 2';
} else if (a === 0 && b === 0) {
  result = 'both a and b are zeros';
} else if (a === b) {
  result = 'a and b are equal';
} else {
  result = 'I give up';
}
```

You can also nest conditions by putting new conditions within any of the blocks:

```
if (a === 1) {
  if (b === 2) {
    result = 'a is 1 and b is 2';
  } else {
    result = 'a is 1 but b is definitely not 2';
  }
} else {
  result = 'a is not 1, no idea about b';
}
```

Code blocks

In the preceding examples, you saw the use of code blocks. Let's take a moment to clarify what a block of code is, because you use blocks extensively when constructing conditions and loops.

A block of code consists of zero or more expressions enclosed in curly brackets:

```
{
  var a = 1;
  var b = 3;
}
```

You can nest blocks within each other indefinitely:

```
{
  var a = 1;
  var b = 3;
  var c, d;
  {
    c = a + b;
```

```
      {
        d = a - b;
      }
    }
  }
```

Best practice tips

- Use end-of-line semicolons, as discussed previously in the chapter. Although the semicolon is optional when you have only one expression per line, it's good to develop the habit of using them. For best readability, the individual expressions inside a block should be placed one per line and separated by semicolons.
- Indent any code placed within curly brackets. Some programmers like one tab indentation, some use four spaces, and some use two spaces. It really doesn't matter, as long as you're consistent. In the preceding example, the outer block is indented with two spaces, the code in the first nested block is indented with four spaces, and the innermost block is indented with six spaces.
- Use curly brackets. When a block consists of only one expression, the curly brackets are optional, but for readability and maintainability, you should get into the habit of always using them, even when they're optional.

Checking if a variable exists

Let's apply the new knowledge about conditions for something practical. It's often necessary to check whether a variable exists. The laziest way to do this is to simply put the variable in the condition part of the `if`, for example, `if (somevar) {...}`. But, this is not necessarily the best method. Let's take a look at an example that tests whether a variable called `somevar` exists, and if so, sets the `result` variable to `yes`:

```
> var result = '';
> if (somevar) {
    result = 'yes';
  }
ReferenceError: somevar is not defined

> result;
""
```

This code obviously works because the end result was not "yes". But firstly, the code generated an error: `somevar` is not defined, and you don't want your code to behave like that. Secondly, just because `if (somevar)` returns `false` doesn't mean that `somevar` is not defined. It could be that `somevar` is defined and initialized but contains a falsy value like `false` or `0`.

A better way to check if a variable is defined is to use `typeof`:

```
> var result = "";
> if (typeof somevar !== "undefined") {
    result = "yes";
  }
> result;
""
```

`typeof` always returns a string, and you can compare this string with the string `"undefined"`. Note that the variable `somevar` may have been declared but not assigned a value yet, and you'll still get the same result. So, when testing with `typeof` like this, you're really testing whether the variable has any value other than the value `undefined`:

```
> var somevar;
> if (typeof somevar !== "undefined") {
    result = "yes";
  }
> result;
""

> somevar = undefined;
> if (typeof somevar !== "undefined") {
    result = "yes";
  }
> result;
""
```

If a variable is defined and initialized with any value other than `undefined`, its type returned by `typeof` is no longer `"undefined"`:

```
> somevar = 123;
> if (typeof somevar !== "undefined") {
    result = 'yes';
  }
> result;
"yes"
```

Alternative if syntax

When you have a simple condition, you can consider using an alternative `if` syntax. Take a look at this:

```
var a = 1;
var result = '';
if (a === 1) {
  result = "a is one";
} else {
  result = "a is not one";
}
```

You can also write this as:

```
> var a = 1;
> var result = (a === 1) ? "a is one" : "a is not one";
```

You should only use this syntax for simple conditions. Be careful not to abuse it, as it can easily make your code unreadable. Here's an example.

Let's say you want to make sure a number is within a certain range, say between 50 and 100:

```
> var a = 123;
> a = a > 100 ? 100 : a < 50 ? 50: a;
> a;
100
```

It may not be clear how this code works exactly because of the multiple `?`. Adding parentheses makes it a little clearer:

```
> var a = 123;
> a = (a > 100 ? 100 : a < 50) ? 50 : a;
> a;
50
```

```
> var a = 123;
> a = a > 100 ? 100 : (a < 50 ? 50 : a);
> a;
100
```

`?:` is called a ternary operator because it takes three operands.

Switch

If you find yourself using an `if` condition and having too many `else if` parts, you could consider changing the `if` to a `switch`:

```
var a = '1',
    result = '';
switch (a) {
case 1:
  result = 'Number 1';
  break;
case '1':
  result = 'String 1';
  break;
default:
  result = 'I don\'t know';
  break;
}
```

The result after executing this is `"String 1"`. Let's see what the parts of a switch are:

- The `switch` statement.
- An expression in parentheses. The expression most often contains a variable, but can be anything that returns a value.
- A number of `case` blocks enclosed in curly brackets.
- Each `case` statement is followed by an expression. The result of the expression is compared to the expression found after the `switch` statement. If the result of the comparison is `true`, the code that follows the colon after the case is executed.
- There is an optional `break` statement to signal the end of the `case` block. If this `break` statement is reached, the `switch` is all done. Otherwise, if the `break` is missing, the program execution enters the next `case` block.
- There's an optional default case marked with the `default` statement and followed by a block of code. The default case is executed if none of the previous cases evaluated to `true`.

In other words, the step-by-step procedure for executing a `switch` statement is as follows:

1. Evaluate the `switch` expression found in parentheses; remember it.
2. Move to the first `case` and compare its value with the one from step 1.
3. If the comparison in step 2 returns `true`, execute the code in the `case` block.
4. After the case block is executed, if there's a `break` statement at the end of it, exit the switch.
5. If there's no `break` or step 2 returned `false`, move on to the next `case` block.
6. Repeat steps 2 to 5.
7. If you are still here (no exit in step 4), execute the code following the `default` statement.

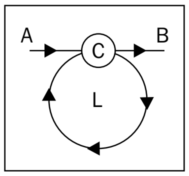
Best practice tips

- Indent the code that follows the `case` lines. You can also indent `case` from the `switch`, but that doesn't give you much in terms of readability.
- Don't forget to `break`.
- Sometimes, you may want to omit the `break` intentionally, but that's rare. It's called a fall-through and should always be documented because it may look like an accidental omission. On the other hand, sometimes you may want to omit the whole code block following a `case` and have two cases sharing the same code. This is fine, but doesn't change the rule that if there's code that follows a `case` statement, this code should end with a `break`. In terms of indentation, aligning the `break` with the `case` or with the code inside the case is a personal preference; again, being consistent is what matters.
- Use the `default` case. This helps you make sure you always have a meaningful result after the `switch` statement, even if none of the cases matches the value being switched.

Loops

The `if-else` and `switch` statements allow your code to take different paths, as if you're at a crossroad and decide which way to go depending on a condition. Loops, on the other hand, allow your code to take a few roundabouts before merging back into the main road. How many repetitions? That depends on the result of evaluating a condition before (or after) each iteration.

Let's say you are (your program execution is) traveling from A to B. At some point, you reach a place where you evaluate a condition, C. The result of evaluating C tells you if you should go into a loop, L. You make one iteration and arrive at C again. Then, you evaluate the condition once again to see if another iteration is needed. Eventually, you move on your way to B.



An infinite loop is when the condition is always `true` and your code gets stuck in the loop "forever". This is, of course, a logical error, and you should look out for such scenarios.

In JavaScript, there are four types of loops:

- `while` loops
- `do-while` loops
- `for` loops
- `for-in` loops

While loops

`while` loops are the simplest type of iteration. They look like this:

```
var i = 0;
while (i < 10) {
  i++;
}
```

The `while` statement is followed by a condition in parentheses and a code block in curly brackets. As long as the condition evaluates to `true`, the code block is executed over and over again.

Do-while loops

`do-while` loops are a slight variation of `while` loops. An example is shown as follows:

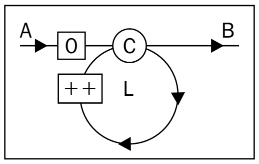
```
var i = 0;
do {
  i++;
} while (i < 10);
```

Here, the `do` statement is followed by a code block and a condition after the block. This means that the code block is always executed, at least once, before the condition is evaluated.

If you initialize `i` to `11` instead of `0` in the last two examples, the code block in the first example (the `while` loop) will not be executed, and `i` will still be `11` at the end, while in the second (the `do-while` loop), the code block will be executed once and `i` will become `12`.

For loops

`for` is the most widely used type of loop, and you should make sure you're comfortable with this one. It requires just a little bit more in terms of syntax.



In addition to the condition C and the code block L, you have the following:

- Initialization—code that is executed before you even enter the loop (marked with `0` in the diagram)
- Increment—code that is executed after every iteration (marked with `++` in the diagram)

The most widely used `for` loop pattern is:

- In the initialization part, you define a variable (or set the initial value of an existing variable), most often called `i`
- In the condition part, you compare `i` to a boundary value, like `i < 100`
- In the increment part, you increase `i` by `1`, like `i++`

Here's an example:

```
var punishment = '';
for (var i = 0; i < 100; i++) {
  punishment += 'I will never do this again, ';
}
```

All three parts (initialization, condition, and increment) can contain multiple expressions separated by commas. Say you want to rewrite the example and define the variable `punishment` inside the initialization part of the loop:

```
for (var i = 0, punishment = ''; i < 100; i++) {
  punishment += 'I will never do this again, ';
}
```

Can you move the body of the loop inside the increment part? Yes, you can, especially as it's a one-liner. This gives you a loop that looks a little awkward, as it has no body. Note that this is just an intellectual exercise; it's not recommended that you write awkward-looking code:

```
for (
  var i = 0, punishment = '';
  i < 100;
  i++, punishment += 'I will never do this again, ') {

  // nothing here

}
```

These three parts are all optional. Here's another way of rewriting the same example:

```
var i = 0, punishment = '';
for (;;) {
  punishment += 'I will never do this again, ';
  if (++i == 100) {
    break;
  }
}
```

Although the last rewrite works exactly the same way as the original, it's longer and harder to read. It's also possible to achieve the same result by using a `while` loop. But, `for` loops make the code tighter and more robust because the mere syntax of the `for` loop makes you think about the three parts (initialization, condition, and increment), and thus helps you reconfirm your logic and avoid situations such as being stuck in an infinite loop.

The `for` loops can be nested within each other. Here's an example of a loop that is nested inside another loop and assembles a string containing 10 rows and 10 columns of asterisks. Think of `i` being the row and `j` being the column of an "image":

```
var res = '\n';
for (var i = 0; i < 10; i++) {
  for (var j = 0; j < 10; j++) {
    res += '* ';
  }
  res += '\n';
}
```

The result is a string like the following:

```
"
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
"
```

Here's another example that uses nested loops and a modulo operation to draw a snowflake-like result:

```
var res = '\n', i, j;
for (i = 1; i <= 7; i++) {
  for (j = 1; j <= 15; j++) {
    res += (i * j) % 8 ? ' ' : '*';
  }
  res += '\n';
}
```

The result is:

```
"
   *
 * * *
   *
*******
   *
 * * *
   *
"
```

For-in loops

The `for-in` loop is used to iterate over the elements of an array (or an object, as you'll see later). This is its only use; it cannot be used as a general-purpose repetition mechanism that replaces `for` or `while`. Let's see an example of using a `for-in` to loop through the elements of an array. But, bear in mind that this is for informational purposes only, as `for-in` is mostly suitable for objects, and the regular `for` loop should be used for arrays.

In this example, you iterate over all of the elements of an array and print out the index (the key) and the value of each element:

```
// example for information only
// for-in loops are used for objects
// regular for is better suited for arrays

var a = ['a', 'b', 'c', 'x', 'y', 'z'];

var result = '\n';

for (var i in a) {
  result += 'index: ' + i + ', value: ' + a[i] + '\n';
}
```

The result is:

```
"
index: 0, value: a
index: 1, value: b
index: 2, value: c
index: 3, value: x
index: 4, value: y
index: 5, value: z
"
```

Comments

One last thing for this chapter: comments. Inside your JavaScript program, you can put comments. These are ignored by the JavaScript engine and don't have any effect on how the program works. But, they can be invaluable when you revisit your code after a few months, or transfer the code to someone else for maintenance.

Two types of comments are allowed:

- Single line comments start with `//` and end at the end of the line.
- Multiline comments start with `/*` and end with `*/` on the same line or any subsequent line. Note that any code in between the comment start and the comment end is ignored.

Some examples are as follows:

```
// beginning of line

var a = 1; // anywhere on the line

/* multi-line comment on a single line */

/*
  comment that spans several lines
*/
```

There are even utilities, such as JSDoc and YUIDoc, that can parse your code and extract meaningful documentation based on your comments.

Summary

In this chapter, you learned a lot about the basic building blocks of a JavaScript program. Now you know the primitive data types:

- Number
- String
- Boolean
- Undefined
- Null

You also know quite a few operators:

- Arithmetic operators: `+`, `-`, `*`, `/`, and `%`
- Increment operators: `++` and `--`
- Assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, and `%=`
- Special operators: `typeof` and `delete`
- Logical operators: `&&`, `||`, and `!`
- Comparison operators: `==`, `===`, `!=`, `!==`, `<`, `>`, `>=`, and `<=`
- The ternary operator `?:`

Then, you learned how to use arrays to store and access data, and finally you saw different ways to control the flow of your program—using conditions (`if-else` or `switch`) and loops (`while`, `do-while`, `for`, and `for-in`).

This is quite a bit of information; now take a moment to go through the exercises below, then give yourself a well-deserved pat on the back before diving into the next chapter. More fun is coming up!

Exercises

1. What is the result of executing each of these lines in the console? Why?

```
> var a; typeof a;
> var s = '1s'; s++;
> !!"false";
> !!undefined;
> typeof -Infinity;
> 10 % "0";
> undefined == null;
> false === "";
> typeof "2E+2";
> a = 3e+3; a++;
```

2. What is the value of `v` after the following?

```
> var v = v || 10;
```

   Experiment by first setting `v` to `100`, `0`, or `null`.

3. Write a small program that prints out the multiplication table. Hint: use a loop nested inside another loop.

3 Functions

Mastering functions is an important skill when you learn any programming language, and even more so when it comes to JavaScript. This is because JavaScript has many uses for functions, and much of the language's flexibility and expressiveness comes from them. Where most programming languages have a special syntax for some object-oriented features, JavaScript just uses functions. This chapter will cover:

- How to define and use a function
- Passing arguments to a function
- Predefined functions that are available to you "for free"
- The scope of variables in JavaScript
- The concept that functions are just data, albeit a special type of data

Understanding these topics will provide a solid base that will allow you to dive into the second part of the chapter, which shows some interesting applications of functions:

- Using anonymous functions
- Callbacks
- Immediate (self-invoking) functions
- Inner functions (functions defined inside other functions)
- Functions that return functions
- Functions that redefine themselves
- Closures

What is a function?

Functions allow you to group together some code, give this code a name, and reuse it later, addressing it by the name you gave it. Let's see an example:

```
function sum(a, b) {
  var c = a + b;
  return c;
}
```

The parts that make up a function are shown as follows:

- The `function` statement.
- The name of the function, in this case `sum`.
- The function parameters, in this case `a` and `b`. A function can take any number of parameters, separated by commas.
- A code block, also called the body of the function.
- The `return` statement. A function always returns a value. If it doesn't return a value explicitly, it implicitly returns the value `undefined`.

Note that a function can only return a single value. If you need to return more values, you can simply return an array that contains all of the values you need as elements of this array.

The preceding syntax is called a function declaration. It's just one of the ways to create a function in JavaScript, and more ways are coming up.

Calling a function

In order to make use of a function, you need to call it. You call a function simply by using its name optionally followed by any number of values in parentheses. "To invoke" a function is another way of saying "to call".

Let's call the function `sum()`, passing two arguments and assigning the value that the function returns to the variable `result`:

```
> var result = sum(1, 2);
> result;
3
```

Parameters

When defining a function, you can specify what parameters the function expects to receive when it's called. A function may not require any parameters, but if it does and you forget to pass them, JavaScript will assign the value `undefined` to the ones you skipped. In the next example, the function call returns `NaN` because it tries to sum `1` and `undefined`:

```
> sum(1);
NaN
```

Technically speaking, there is a difference between parameters and arguments, although the two are often used interchangeably. Parameters are defined together with the function, while arguments are passed to the function when it's called. Consider this:

```
> function sum(a, b) {
    return a + b;
  }
> sum(1, 2);
```

Here, `a` and `b` are parameters, while `1` and `2` are arguments.

JavaScript is not picky at all when it comes to accepting arguments. If you pass more than the function expects, the extra ones will be silently ignored:

```
> sum(1, 2, 3, 4, 5);
3
```

What's more, you can create functions that are flexible about the number of parameters they accept. This is possible thanks to the special value `arguments` that is created automatically inside each function. Here's a function that simply returns whatever arguments are passed to it:

```
> function args() {
    return arguments;
  }
> args();
[]

> args( 1, 2, 3, 4, true, 'ninja');
[1, 2, 3, 4, true, "ninja"]
```

By using `arguments`, you can improve the `sum()` function to accept any number of arguments and add them all up:

```
function sumOnSteroids() {
  var i,
      res = 0,
      number_of_params = arguments.length;
  for (i = 0; i < number_of_params; i++) {
    res += arguments[i];
  }
  return res;
}
```

If you test this function by calling it with a different number of arguments (or even none at all), you can verify that it works as expected:

```
> sumOnSteroids(1, 1, 1);
3

> sumOnSteroids(1, 2, 3, 4);
10

> sumOnSteroids(1, 2, 3, 4, 4, 3, 2, 1);
20

> sumOnSteroids(5);
5

> sumOnSteroids();
0
```

The expression `arguments.length` returns the number of arguments passed when the function was called. Don't worry if the syntax is unfamiliar, we'll examine it in detail in the next chapter. You'll also see that `arguments` is not an array (although it sure looks like one), but an array-like object.

Predefined functions

There are a number of functions that are built into the JavaScript engine and are available for you to use. Let's take a look at them. While doing so, you'll have a chance to experiment with functions, their arguments and return values, and become comfortable working with functions. Following is a list of the built-in functions:

- `parseInt()`
- `parseFloat()`
- `isNaN()`

- `isFinite()`
- `encodeURI()`
- `decodeURI()`
- `encodeURIComponent()`
- `decodeURIComponent()`
- `eval()`

The black box function

Often, when you invoke functions, your program doesn't need to know how these functions work internally. You can think of a function as a black box: you give it some values (as input arguments) and then you take the output result it returns. This is true for any function—one that's built into the JavaScript engine, one that you create, or one that a co-worker or someone else created.

parseInt()

`parseInt()` takes any type of input (most often a string) and tries to make an integer out of it. If it fails, it returns `NaN`:

```
> parseInt('123');
123

> parseInt('abc123');
NaN

> parseInt('1abc23');
1

> parseInt('123abc');
123
```

The function accepts an optional second parameter, which is the **radix**, telling the function what type of number to expect—decimal, hexadecimal, binary, and so on. For example, trying to extract a decimal number out of the string `FF` makes no sense, so the result is **NaN**, but if you try `FF` as a hexadecimal, then you get **255**:

```
> parseInt('FF', 10);
NaN

> parseInt('FF', 16);
255
```

Another example would be parsing a string with a base `10` (decimal) and base `8` (octal):

```
> parseInt('0377', 10);
377

> parseInt('0377', 8);
255
```

If you omit the second argument when calling `parseInt()`, the function will assume `10` (a decimal), with these exceptions:

- If you pass a string beginning with `0x`, then the radix is assumed to be `16` (a hexadecimal number is assumed).
- If the string you pass starts with `0`, the function assumes radix `8` (an octal number is assumed). Consider the following examples:

  ```
  > parseInt('377');
  377

  > parseInt('0377');
  255

  > parseInt('0x377');
  887
  ```

The safest thing to do is to always specify the radix. If you omit the radix, your code will probably still work in 99 percent of cases (because most often you parse decimals), but every once in a while it might cause you a bit of hair loss while debugging some edge cases. For example, imagine you have a form field that accepts calendar days or months and the user types `06` or `08`.

[ECMAScript 5 removes the octal literal values and avoids the confusion with `parseInt()` and unspecified radix.]

parseFloat()

`parseFloat()` is similar to `parseInt()`, but it also looks for decimals when trying to figure out a number from your input. This function takes only one parameter:

```
> parseFloat('123');
123

> parseFloat('1.23');
1.23
```

```
> parseFloat('1.23abc.00');
1.23

> parseFloat('a.bc1.23');
NaN
```

As with `parseInt()`, `parseFloat()` gives up at the first occurrence of an unexpected character, even though the rest of the string might have usable numbers in it:

```
> parseFloat('a123.34');
NaN

> parseFloat('12a3.34');
12
```

`parseFloat()` understands exponents in the input (unlike `parseInt()`):

```
> parseFloat('123e-2');
1.23

> parseFloat('1e10');
10000000000

> parseInt('1e10');
1
```

isNaN()

Using `isNaN()`, you can check if an input value is a valid number that can safely be used in arithmetic operations. This function is also a convenient way to check whether `parseInt()` or `parseFloat()` (or any arithmetic operation) succeeded:

```
> isNaN(NaN);
true

> isNaN(123);
false

> isNaN(1.23);
false

> isNaN(parseInt('abc123'));
true
```

The function will also try to convert the input to a number:

```
> isNaN('1.23');
false

> isNaN('a1.23');
true
```

The `isNaN()` function is useful because the special value `NaN` is not equal to anything including itself. In other words, `NaN === NaN` is `false`. So, `NaN` cannot be used to check if a value is a valid number.

isFinite()

`isFinite()` checks whether the input is a number that is neither `Infinity` nor `NaN`:

```
> isFinite(Infinity);
false

> isFinite(-Infinity);
false

> isFinite(12);
true

> isFinite(1e308);
true

> isFinite(1e309);
false
```

If you are wondering about the results returned by the last two calls, remember from the previous chapter that the biggest number in JavaScript is `1.7976931348623157e+308`, so `1e309` is effectively infinity.

Encode/decode URIs

In a **Uniform Resource Locator** (**URL**) or a **Uniform Resource Identifier** (**URI**), some characters have special meanings. If you want to "escape" those characters, you can use the functions `encodeURI()` or `encodeURIComponent()`. The first one will return a usable URL, while the second one assumes you're only passing a part of the URL, such as a query string for example, and will encode all applicable characters:

```
> var url = 'http://www.packtpub.com/script.php?q=this and that';
> encodeURI(url);
http://www.packtpub.com/script.php?q=this%20and%20that
```

```
> encodeURIComponent(url);
http%3A%2F%2Fwww.packtpub.com%2Fscript.php%3Fq%3Dthis%20and%20that
```

The opposites of `encodeURI()` and `encodeURIComponent()` are `decodeURI()` and `decodeURIComponent()` respectively.

Sometimes, in legacy code, you might see the functions `escape()` and `unescape()` used to encode and decode URLs, but these functions have been deprecated; they encode differently and should not be used.

eval()

`eval()` takes a string input and executes it as a JavaScript code:

```
> eval('var ii = 2;');
> ii;
2
```

So, `eval('var ii = 2;')` is the same as `var ii = 2;`.

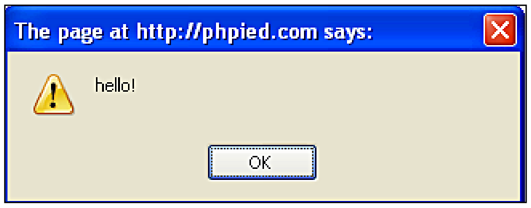
`eval()` can be useful sometimes, but should be avoided if there are other options. Most of the time there are alternatives, and in most cases the alternatives are more elegant and easier to write and maintain. "Eval is evil" is a mantra you can often hear from seasoned JavaScript programmers. The drawbacks of using `eval()` are:

- **Security** – JavaScript is powerful, which also means it can cause damage. If you don't trust the source of the input you pass to `eval()`, just don't use it.
- **Performance** – It's slower to evaluate "live" code than to have the code directly in the script.

A bonus – the alert() function

Let's take a look at another common function—`alert()`. It's not part of the core JavaScript (it's nowhere to be found in the ECMA specification), but it's provided by the host environment—the browser. It shows a string of text in a message box. It can also be used as a primitive debugging tool, although the debuggers in modern browsers are much better suited for this purpose.

Here's a screenshot showing the result of executing the code `alert("hello!")`:



The page at http://phpied.com says:

hello!

Before using this function, bear in mind that it blocks the browser thread, meaning that no other code will be executed until the user closes the alert. If you have a busy Ajax-type application, it's generally not a good idea to use `alert()`.

Scope of variables

It's important to note, especially if you have come to JavaScript from another language, that variables in JavaScript are not defined in a block scope, but in a function scope. This means that if a variable is defined inside a function, it's not visible outside of the function. However, if it's defined inside an `if` or a `for` code block, it's visible outside the block. The term "global variables" describes variables you define outside of any function (in the global program code), as opposed to "local variables", which are defined inside a function. The code inside a function has access to all global variables as well as to its own local ones.

In the next example:

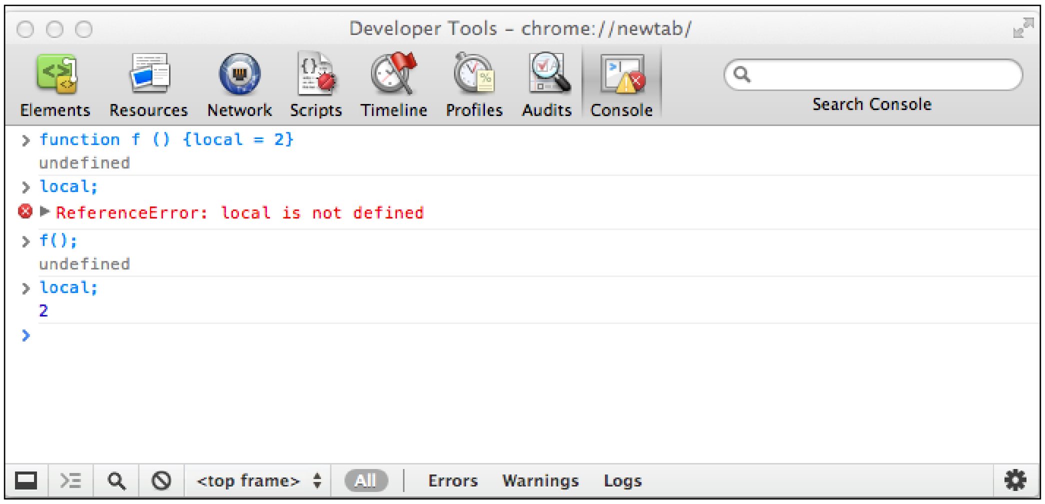
- The `f()` function has access to the `global` variable
- Outside the `f()` function, the `local` variable doesn't exist

```
var global = 1;
function f() {
  var local = 2;
  global++;
  return global;
}
```

Let's test this:

```
> f();
2

> f();
3

> local;
ReferenceError: local is not defined
```

It's also important to note that if you don't use `var` to declare a variable, this variable is automatically assigned a global scope. Let's see an example:



```
> function f () {local = 2}
  undefined
> local;
✖ ▶ ReferenceError: local is not defined
> f();
  undefined
> local;
  2
>
```

What happened? The function `f()` contains the variable `local`. Before calling the function, the variable doesn't exist. When you call the function for the first time, the variable `local` is created with a global scope. Then, if you access `local` outside the function, it will be available.

Best practice tips

- Minimize the number of global variables in order to avoid naming collisions. Imagine two people working on two different functions in the same script, and they both decide to use the same name for their global variable. This could easily lead to unexpected results and hard-to-find bugs.
- Always declare your variables with the `var` statement.
- Consider a "single var" pattern. Define all variables needed in your function at the very top of the function so you have a single place to look for variables and hopefully prevent accidental globals.

Variable hoisting

Here's an interesting example that shows an important aspect of local versus global scoping:

```
var a = 123;

function f() {
  alert(a);
  var a = 1;
  alert(a);
}

f();
```

You might expect that the first `alert()` will display **123** (the value of the global variable `a`) and the second will display **1** (the local variable `a`). But, this is not the case. The first alert will show **undefined**. This is because inside the function the local scope is more important than the global scope. So, a local variable overwrites any global variable with the same name. At the time of the first `alert()`, the variable `a` was not yet defined (hence the value `undefined`), but it still existed in the local space due to the special behavior called **hoisting**.

When your JavaScript program execution enters a new function, all the variables declared anywhere in the function are moved (or elevated, or hoisted) to the top of the function. This is an important concept to keep in mind. Further, only the declaration is hoisted, meaning only the presence of the variable is moved to the top. Any assignments stay where they are. In the preceding example, the declaration of the local variable `a` was hoisted to the top. Only the declaration was hoisted, but not the assignment to 1. It's as if the function was written like this:

```
var a = 123;

function f() {
  var a; // same as: var a = undefined;
  alert(a); // undefined
  a = 1;
  alert(a); // 1
}
```

You can also adopt the single var pattern mentioned previously in the best practice section. In this case, you'll be doing a sort of manual variable hoisting to prevent confusion with the JavaScript hoisting behavior.

Functions are data

Functions in JavaScript are actually data. This is an important concept that we'll need later on. This means that you can create a function and assign it to a variable:

```
var f = function () {
  return 1;
};
```

This way of defining a function is sometimes referred to as **function literal notation**.

The part `function () { return 1; }` is a **function expression**. A function expression can optionally have a name, in which case it becomes a **named function expression** (**NFE**). So, this is also allowed, although rarely seen in practice (and causes IE to mistakenly create two variables in the enclosing scope: `f` and `myFunc`):

```
var f = function myFunc() {
  return 1;
};
```

As you can see, there's no difference between a named function expression and a function declaration. But they are, in fact, different. The only way to distinguish between the two is to look at the context in which they are used. Function declarations may only appear in program code (in a body of another function or in the main program). You'll see many more examples of functions later on in the book that will clarify these concepts.

When you use the `typeof` operator on a variable that holds a function value, it returns the string `"function"`:

```
> function define() {
    return 1;
  }

> var express = function () {
    return 1;
  };

> typeof define;
"function"


> typeof express;
"function"
```

So, JavaScript functions are data, but a special kind of data with two important features:

- They contain code
- They are executable (they can be invoked)

As you have seen before, the way to execute a function is by adding parentheses after its name. As the next example demonstrates, this works regardless of how the function was defined. In the example, you can also see how a function is treated as a regular value: it can be copied to a different variable:

```
> var sum = function (a, b) {
    return a + b;
  };

> var add = sum;
> typeof add;
function

> add(1, 2);
3
```

Because functions are data assigned to variables, the same rules for naming functions apply as for naming variables—a function name cannot start with a number and it can contain any combination of letters, numbers, the underscore character, and the dollar sign.

Anonymous functions

As you now know, there exists a function expression syntax where you can have a function defined like this:

```
var f = function (a) {
  return a;
};
```

This is also often called an anonymous function (as it doesn't have a name), especially when such a function expression is used even without assigning it to a variable. In this case, there can be two elegant uses for such anonymous functions:

- You can pass an anonymous function as a parameter to another function. The receiving function can do something useful with the function that you pass.
- You can define an anonymous function and execute it right away.

Let's see these two applications of anonymous functions in more detail.

Callback functions

Because a function is just like any other data assigned to a variable, it can be defined, copied, and also passed as an argument to other functions.

Here's an example of a function that accepts two functions as parameters, executes them, and returns the sum of what each of them returns:

```
function invokeAdd(a, b) {
  return a() + b();
}
```

Now let's define two simple additional functions (using a function declaration pattern) that only return hardcoded values:

```
function one() {
  return 1;
}

function two() {
  return 2;
}
```

Now you can pass those functions to the original function, `invokeAdd()`, and get the result:

```
> invokeAdd(one, two);
3
```

Another example of passing a function as a parameter is to use anonymous functions (function expressions). Instead of defining `one()` and `two()`, you can simply do the following:

```
> invokeAdd(function () {return 1; }, function () {return 2; });
3
```

Or, you can make it more readable as shown in the following code:

```
> invokeAdd(
    function () { return 1; },
    function () { return 2; }
  );
3
```

Or, you can do the following:

```
> invokeAdd(
    function () {
      return 1;
    },
    function () {
      return 2;
    }
  );
3
```

When you pass a function, A, to another function, B, and then B executes A, it's often said that A is a **callback** function. If A doesn't have a name, then you can say that it's an anonymous callback function.

When are callback functions useful? Let's see some examples that demonstrate the benefits of callback functions, namely:

- They let you pass functions without the need to name them (which means there are fewer variables floating around)
- You can delegate the responsibility of calling a function to another function (which means there is less code to write)
- They can help with performance

Callback examples

Take a look at this common scenario: you have a function that returns a value, which you then pass to another function. In our example, the first function, `multiplyByTwo()`, accepts three parameters, loops through them, multiplies them by two, and returns an array containing the result. The second function, `addOne()`, takes a value, adds one to it, and returns it:

```
function multiplyByTwo(a, b, c) {
  var i, ar = [];
  for (i = 0; i < 3; i++) {
    ar[i] = arguments[i] * 2;
  }
  return ar;
}

function addOne(a) {
  return a + 1;
}
```

Let's test these functions:

```
> multiplyByTwo(1, 2, 3);
[2, 4, 6]

> addOne(100);
101
```

Now let's say you want to have an array, `myarr`, that contains three elements, and each of the elements is to be passed through both functions. First, let's start with a call to `multiplyByTwo()`:

```
> var myarr = [];
> myarr = multiplyByTwo(10, 20, 30);
[20, 40, 60]
```

Now loop through each element, passing it to `addOne()`:

```
> for (var i = 0; i < 3; i++) {
    myarr[i] = addOne(myarr[i]);
  }
> myarr;
[21, 41, 61]
```

As you can see, everything works fine, but there's room for improvement. For example: there were two loops. Loops can be expensive if they go through a lot of repetitions. You can achieve the same result with only one loop. Here's how to modify `multiplyByTwo()` so that it accepts a callback function and invokes that callback on every iteration:

```
function multiplyByTwo(a, b, c, callback) {
  var i, ar = [];
  for (i = 0; i < 3; i++) {
    ar[i] = callback(arguments[i] * 2);
  }
  return ar;
}
```

By using the modified function, all the work is done with just one function call, which passes the start values and the `callback` function:

```
> myarr = multiplyByTwo(1, 2, 3, addOne);
[3, 5, 7]
```

Instead of defining `addOne()`, you can use an anonymous function, therefore saving an extra global variable:

```
> multiplyByTwo(1, 2, 3, function (a) {
    return a + 1;
  });
[3, 5, 7]
```

Anonymous functions are easy to change should the need arise:

```
> multiplyByTwo(1, 2, 3, function (a) {
    return a + 2;
  });
[4, 6, 8]
```

Immediate functions

So far, we have discussed using anonymous functions as callbacks. Let's see another application of an anonymous function: calling a function immediately after it's defined. Here's an example:

```
(
  function () {
    alert('boo');
  }
)();
```

The syntax may look a little scary at first, but all you do is simply place a function expression inside parentheses followed by another set of parentheses. The second set says "execute now" and is also the place to put any arguments that your anonymous function might accept:

```
(
  function (name) {
    alert('Hello ' + name + '!');
  }
)('dude');
```

Alternatively, you can move the closing of the first set of parentheses to the end. Both of these work:

```
(function () {
  // ...
}());
```

```
// vs.

(function () {
  // ...
})();
```

One good application of immediate (self-invoking) anonymous functions is when you want to have some work done without creating extra global variables. A drawback, of course, is that you cannot execute the same function twice. This makes immediate functions best suited for one-off or initialization tasks.

An immediate function can also optionally return a value if you need one. It's not uncommon to see code that looks like the following:

```
var result = (function () {
  // something complex with
  // temporary local variables...
  // ...

  // return something;
}());
```

In this case, you don't need to wrap the function expression in parentheses, you only need the parentheses that invoke the function. So, the following also works:

```
var result = function () {
  // something complex with
  // temporary local variables
  // return something;
}();
```

This syntax works, but may look slightly confusing: without reading the end of the function, you don't know if `result` is a function or the return value of the immediate function.

Inner (private) functions

Bearing in mind that a function is just like any other value, there's nothing that stops you from defining a function inside another function:

```
function outer(param) {
  function inner(theinput) {
    return theinput * 2;
  }
  return 'The result is ' + inner(param);
}
```

Using a function expression, this can also be written as:

```
var outer = function (param) {
  var inner = function (theinput) {
    return theinput * 2;
  };
  return 'The result is ' + inner(param);
};
```

When you call the global function `outer()`, it will internally call the local function `inner()`. Since `inner()` is local, it's not accessible outside `outer()`, so you can say it's a private function:

```
> outer(2);
"The result is 4"

> outer(8);
"The result is 16"

> inner(2);
ReferenceError: inner is not defined
```

The benefits of using private functions are as follows:

- You keep the global namespace clean (less likely to cause naming collisions)
- Privacy—you expose only the functions you decide to the "outside world", keeping to yourself functionality that is not meant to be consumed by the rest of the application

Functions that return functions

As mentioned earlier, a function always returns a value, and if it doesn't do it explicitly with `return`, then it does so implicitly by returning `undefined`. A function can return only one value, and this value can just as easily be another function:

```
function a() {
  alert('A!');
  return function () {
    alert('B!');
  };
}
```

In this example, the function `a()` does its job (says **A!**) and returns another function that does something else (says **B!**). You can assign the return value to a variable and then use this variable as a normal function:

```
> var newFunc = a();
> newFunc();
```

Here, the first line will alert **A!** and the second will alert **B!**.

If you want to execute the returned function immediately without assigning it to a new variable, you can simply use another set of parentheses. The end result will be the same:

```
> a()();
```

Function, rewrite thyself!

Because a function can return a function, you can use the new function to replace the old one. Continuing with the previous example, you can take the value returned by the call to `a()` to overwrite the actual `a()` function:

```
> a = a();
```

The above alerts **A!**, but the next time you call `a()` it alerts **B!**. This is useful when a function has some initial one-off work to do. The function overwrites itself after the first call in order to avoid doing unnecessary repetitive work every time it's called.

In the preceding example, the function was redefined from the outside—the returned value was assigned back to the function. But, the function can actually rewrite itself from the inside:

```
function a() {
  alert('A!');
  a = function () {
    alert('B!');
  };
}
```

If you call this function for the first time, it will:

- Alert **A!** (consider this as being the one-off preparatory work)
- Redefine the global variable `a`, assigning a new function to it

Every subsequent time that the function is called, it will alert **B!**

Here's another example that combines several of the techniques discussed in the last few sections of this chapter:

```
var a = (function () {

  function someSetup() {
    var setup = 'done';
  }

  function actualWork() {
    alert('Worky-worky');
  }

  someSetup();
  return actualWork;

}());
```

In this example:

- You have private functions: `someSetup()` and `actualWork()`.
- You have an immediate function: an anonymous function that calls itself using the parentheses following its definition.
- The function executes for the first time, calls `someSetup()`, and then returns a reference to the variable `actualWork`, which is a function. Notice that there are no parentheses in the `return` statement, because you're returning a function reference, not the result of invoking this function.
- Because the whole thing starts with `var a =`, the value returned by the self-invoked function is assigned to `a`.

If you want to test your understanding of the topics just discussed, answer the following questions. What will the preceding code alert when:

- It is initially loaded?
- You call `a()` afterwards?

These techniques could be really useful when working in the browser environment. Different browsers can have different ways of achieving the same result. If you know that the browser features won't change between function calls, you can have a function determine the best way to do the work in the current browser, then redefine itself so that the "browser capability detection" is done only once. You'll see concrete examples of this scenario later in this book.

Closures

The rest of the chapter is about closures (what better way to close a chapter?). Closures can be a little hard to grasp initially, so don't feel discouraged if you don't "get it" during the first read. You should go through the rest of the chapter and experiment with the examples on you own, but if you feel you don't fully understand the concept, you can come back to it later when the topics discussed previously in this chapter have had a chance to sink in.

Before moving on to closures, let's first review and expand on the concept of scope in JavaScript.

Scope chain

As you know, in JavaScript, there is no curly braces scope, but there is function scope. A variable defined in a function is not visible outside the function, but a variable defined in a code block (for example an `if` or a `for` loop) is visible outside the block:

```
> var a = 1;
> function f() {
    var b = 1;
    return a;
  }
> f();
1

> b;
ReferenceError: b is not defined
```

The variable `a` is in the global space, while `b` is in the scope of the function `f()`. So:

- Inside `f()`, both `a` and `b` are visible
- Outside `f()`, `a` is visible, but `b` is not

If you define a function `inner()` nested inside `outer()`, `inner()` will have access to variables in its own scope, plus the scope of its "parents". This is known as a scope chain, and the chain can be as long (deep) as you need it to be:

```
var global = 1;
function outer() {
  var outer_local = 2;
  function inner() {
    var inner_local = 3;
    return inner_local + outer_local + global;
  }
  return inner();
}
```

Let's test that `inner()` has access to all variables:

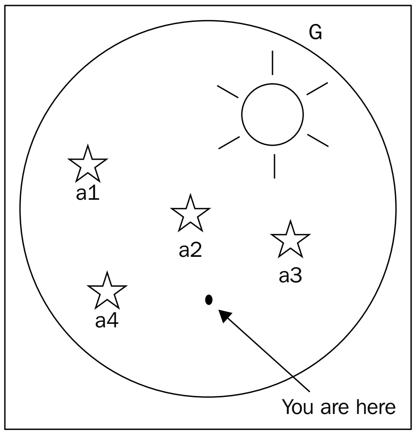
```
  > outer();
  6
```

Breaking the chain with a closure

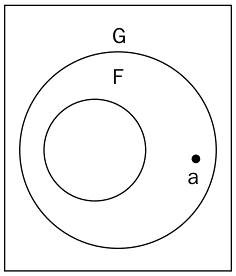
Let's introduce closures with an illustration. Let's look at this code and see what's happening there:

```
  var a = "global variable";
  var F = function () {
    var b = "local variable";
    var N = function () {
      var c = "inner local";
    };
  };
```

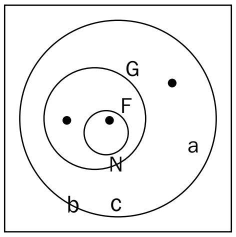
First, there is the global scope `G`. Think of it as the universe, as if it contains everything:



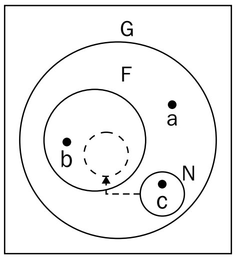
It can contain global variables such as `a1` and `a2` and global functions such as `F`:



Functions have their own private space and can use it to store other variables such as `b` and inner functions such as `N` (for *iNNer*). At some point, you end up with a picture like this:



If you're at point `a`, you're inside the global space. If you're at point `b`, which is inside the space of the function `F`, then you have access to the global space and to the `F` space. If you're at point `c`, which is inside the function `N`, then you can access the global space, the `F` space, and the `N` space. You cannot reach from `a` to `b`, because `b` is invisible outside `F`. But, you can get from `c` to `b` if you want, or from `N` to `b`. The interesting part—the closure effect—happens when somehow `N` breaks out of `F` and ends up in the global space:



What happens then? `N` is in the same global space as `a`. And, as functions remember the environment in which they were defined, `N` will still have access to the `F` space, and hence can access `b`. This is interesting, because `N` is where `a` is and yet `N` does have access to `b`, but `a` doesn't.

And how does `N` break the chain? By making itself global (omitting `var`) or by having `F` deliver (or `return`) it to the global space. Let's see how this is done in practice.

Closure #1

Take a look at this function, which is the same as before, only `F` returns `N` and also `N` returns `b`, to which it has access via the scope chain:

```
var a = "global variable";
var F = function () {
  var b = "local variable";
  var N = function () {
    var c = "inner local";
    return b;
  };
  return N;
};
```

The function `F` contains the variable `b`, which is local, and therefore inaccessible from the global space:

```
> b;
ReferenceError: b is not defined
```

The function `N` has access to its private space, to the `F()` function's space, and to the global space. So, it can see `b`. Since `F()` is callable from the global space (it's a global function), you can call it and assign the returned value to another global variable. The result: a new global function that has access to the `F()` function's private space:

```
> var inner = F();
> inner();
"local variable"
```

Closure #2

The final result of the next example will be the same as the previous example, but the way to achieve it is a little different. `F()` doesn't return a function, but instead it creates a new global function, `inner()`, inside its body.

Let's start by declaring a placeholder for the global function-to-be. This is optional, but it's always good to declare your variables. Then, you can define the function `F()` as follows:

```
var inner; // placeholder
var F = function () {
  var b = "local variable";
  var N = function () {
    return b;
  };
  inner = N;
};
```

Now what happens if you invoke `F()`?:

```
> F();
```

A new function, `N()`,is defined inside `F()` and assigned to the global `inner`. During definition time, `N()` was inside `F()`, so it had access to the `F()` function's scope. `inner()` will keep its access to the `F()` function's scope, even though it's part of the global space:

```
> inner();
"local variable".
```

A definition and closure #3

Every function can be considered a closure. This is because every function maintains a secret link to the environment (the scope) in which it was created. But, most of the time this scope is destroyed unless something interesting happens (as shown above) that causes this scope to be maintained.

Based on what you've seen so far, you can say that a closure is created when a function keeps a link to its parent scope even after the parent has returned. And, every function is a closure because, at the very least, every function maintains access to the global scope, which is never destroyed.

Let's see one more example of a closure, this time using the function parameters. Function parameters behave like local variables to this function, but they are implicitly created (you don't need to use `var` for them). You can create a function that returns another function, which in turn returns its parent's parameter:

```
function F(param) {
  var N = function () {
    return param;
  };
```

```
    param++;
    return N;
  }
```

You use the function like this:

```
  > var inner = F(123);
  > inner();
  124
```

Notice how `param++` was incremented after the function was defined and yet, when called, `inner()` returned the updated value. This demonstrates that the function maintains a reference to the scope where it was defined, not to the variables and their values found in the scope during the function execution.

Closures in a loop

Let's take a look at a canonical rookie mistake when it comes to closures. It can easily lead to hard-to-spot bugs, because on the surface, everything looks normal.

Let's loop three times, each time creating a new function that returns the loop sequence number. The new functions will be added to an array and the array is returned at the end. Here's the function:

```
  function F() {
    var arr = [], i;
    for (i = 0; i < 3; i++) {
      arr[i] = function () {
        return i;
      };
    }
    return arr;
  }
```

Let's run the function, assigning the result to the array `arr`:

```
  > var arr = F();
```

Now you have an array of three functions. Let's invoke them by adding parentheses after each array element. The expected behavior is to see the loop sequence printed out: `0`, `1`, and `2`. Let's try:

```
  > arr[0]();
  3

  > arr[1]();
  3
```

```
> arr[2]();
3
```

Hmm, not quite as expected. What happened here? All three functions point to the same local variable `i`. Why? The functions don't remember values, they only keep a link (reference) to the environment where they were created. In this case, the variable `i` happens to live in the environment where the three functions were defined. So, all functions, when they need to access the value, reach back to the environment and find the most current value of `i`. After the loop, the `i` variable's value is `3`. So, all three functions point to the same value.

Why three and not two is another good question to think about for better understanding the `for` loop.

So, how do you implement the correct behavior? The answer is to use another closure:

```
function F() {
  var arr = [], i;
  for (i = 0; i < 3; i++) {
    arr[i] = (function (x) {
      return function () {
        return x;
      };
    }(i));
  }
  return arr;
}
```

This gives you the expected result:

```
> var arr = F();
> arr[0]();
0

> arr[1]();
1

> arr[2]();
2
```

Here, instead of just creating a function that returns `i`, you pass the `i` variable's current value to another immediate function. In this function, `i` becomes the local value `x`, and `x` has a different value every time.

Alternatively, you can use a "normal" (as opposed to an immediate) inner function to achieve the same result. The key is to use the middle function to "localize" the value of `i` at every iteration:

```
function F() {

  function binder(x) {
    return function () {
      return x;
    };
  }

  var arr = [], i;
  for (i = 0; i < 3; i++) {
    arr[i] = binder(i);
  }
  return arr;
}
```

Getter/setter

Let's see two more examples of using closures. The first one involves the creation of getter and setter functions. Imagine you have a variable that should contain a specific type of values or a specific range of values. You don't want to expose this variable because you don't want just any part of the code to be able to alter its value. You protect this variable inside a function and provide two additional functions: one to get the value and one to set it. The one that sets it could contain some logic to validate a value before assigning it to the protected variable. Let's make the validation part simple (for the sake of keeping the example short) and only accept number values.

You place both the getter and the setter functions inside the same function that contains the `secret` variable so that they share the same scope:

```
var getValue, setValue;

(function () {

  var secret = 0;

  getValue = function () {
    return secret;
  };

  setValue = function (v) {
    if (typeof v === "number") {
      secret = v;
    }
  };

}());
```

In this case, the function that contains everything is an immediate function. It defines `setValue()` and `getValue()` as global functions, while the `secret` variable remains local and inaccessible directly:

```
    > getValue();
    0

    > setValue(123);
    > getValue();
    123

    > setValue(false);
    > getValue();
    123
```

Iterator

The last closure example (also the last example in the chapter) shows the use of a closure to accomplish an iterator functionality.

You already know how to loop through a simple array, but there might be cases where you have a more complicated data structure with different rules as to what the sequence of values has. You can wrap the complicated "who's next" logic into an easy-to-use `next()` function. Then, you simply call `next()` every time you need the consecutive value.

For this example, let's just use a simple array and not a complex data structure. Here's an initialization function that takes an input array and also defines a secret pointer, `i`, that will always point to the next element in the array:

```
    function setup(x) {
      var i = 0;
      return function () {
        return x[i++];
      };
    }
```

Calling the `setup()` function with a data array will create the `next()` function for you:

```
> var next = setup(['a', 'b', 'c']);
```

From there it's easy and fun: calling the same function over and over again gives you the next element:

```
> next();
"a"

> next();
"b"

> next();
"c"
```

Summary

You have now completed the introduction to the fundamental concepts related to functions in JavaScript. You've been laying the groundwork that will allow you to quickly grasp the concepts of object-oriented JavaScript and the patterns used in modern JavaScript programming. So far, we've been avoiding the OO features, but as you have reached this point in the book, it's only going to get more interesting from here on in. Let's take a moment and review the topics discussed in this chapter:

- The basics of how to define and invoke (call) a function using either a function declaration syntax or a function expression
- Function parameters and their flexibility
- Built-in functions—`parseInt()`, `parseFloat()`, `isNaN()`, `isFinite()`, and `eval()`—and the four functions to encode/decode a URL
- The scope of variables in JavaScript—no curly braces scope, variables have only function scope and the scope chain
- Functions as data—a function is like any other piece of data that you assign to a variable and a lot of interesting applications follow from this, such as:
    - Private functions and private variables
    - Anonymous functions
    - Callbacks
    - Immediate functions
    - Functions overwriting themselves
- Closures

Exercises

1. Write a function that converts a hexadecimal color, for example blue (`#0000FF`), into its RGB representation `rgb(0, 0, 255)`. Name your function `getRGB()` and test it with the following code. Hint: treat the string as an array of characters:

   ```
   > var a = getRGB("#00FF00");
   > a;
   ```
   "rgb(0, 255, 0)"

2. What do each of these lines print in the console?

   ```
   > parseInt(1e1);
   > parseInt('1e1');
   > parseFloat('1e1');
   > isFinite(0/10);
   > isFinite(20/0);
   > isNaN(parseInt(NaN));
   ```

3. What does this following code alert?

   ```
   var a = 1;

   function f() {
     function n() {
       alert(a);
     }
     var a = 2;
     n();
   }

   f();
   ```

4. All these examples alert **"Boo!"**. Can you explain why?
   - Example 1:

     ```
     var f = alert;
     eval('f("Boo!")');
     ```

   - Example 2:

     ```
     var e;
     var f = alert;
     eval('e=f')('Boo!');
     ```

   - Example 3:

     ```
     (function(){
       return alert;}
     )()('Boo!');
     ```

4
Objects

Now that you've mastered JavaScript's primitive data types, arrays, and functions, it's time to make true to the promise of the book title and talk about objects.

In this chapter, you will learn:

- How to create and use objects
- What are the constructor functions
- What types of built-in JavaScript objects exist and what they can do for you

From arrays to objects

As you already know from *Chapter 2*, *Primitive Data Types, Arrays, Loops, and Conditions,* an array is just a list of values. Each value has an index (a numeric key) that starts from zero and increments by one for each value.

```
> var myarr = ['red', 'blue', 'yellow', 'purple'];
> myarr;
["red", "blue", "yellow", "purple"].

> myarr[0];
"red"

> myarr[3];
"purple"
```

If you put the indexes in one column and the values in another, you'll end up with a table of key/value pairs shown as follows:

Key	Value
0	red
1	blue
2	yellow
3	purple

An object is similar to an array, but with the difference that you define the keys yourself. You're not limited to using only numeric indexes and you can use friendlier keys, such as `first_name`, `age`, and so on.

Let's take a look at a simple object and examine its parts:

```
var hero = {
  breed: 'Turtle',
  occupation: 'Ninja'
};
```

You can see that:

- The name of the variable that refers to the object is `hero`
- Instead of `[` and `]`, which you use to define an array, you use `{` and `}` for objects
- You separate the elements (called properties) contained in the object with commas
- The key/value pairs are divided by colons, as in `key: value`

The keys (names of the properties) can optionally be placed in quotation marks. For example, these are all the same:

```
var hero = {occupation: 1};
var hero = {"occupation": 1};
var hero = {'occupation': 1};
```

It's recommended that you don't quote the names of the properties (it's less typing), but there are cases when you must use quotes:

- If the property name is one of the reserved words in JavaScript (see *Chapter 9, Reserved Words*)

- If it contains spaces or special characters (anything other than letters, numbers, and the _ and `$` characters)
- If it starts with a number

In other words, if the name you have chosen for a property is not a valid name for a variable in JavaScript, then you need to wrap it in quotes.

Have a look at this bizarre-looking object:

```
var o = {
  $omething: 1,
  'yes or no': 'yes',
  '!@#$%^&*': true
};
```

This is a valid object. The quotes are required for the second and the third properties, otherwise you'll get an error.

Later in this chapter, you'll see other ways to define objects and arrays in addition to `[]` and `{}`. But first, let's introduce this bit of terminology: defining an array with `[]` is called **array literal** notation, and defining an object using the curly braces `{}` is called **object literal** notation.

Elements, properties, methods, and members

When talking about arrays, you say that they contain elements. When talking about objects, you say that they contain properties. There isn't any significant difference in JavaScript; it's just the terminology that people are used to, likely from other programming languages.

A property of an object can point to a function, because functions are just data. Properties that point to functions are also called methods. In the following example, `talk` is a method:

```
var dog = {
  name: 'Benji',
  talk: function () {
    alert('Woof, woof!');
  }
};
```

As you have seen in the previous chapter, it's also possible to store functions as array elements and invoke them, but you'll not see much code like this in practice:

```
> var a = [];
> a[0] = function (what) { alert(what); };
> a[0]('Boo!');
```

You can also see people using the word members to refer to properties of an object, most often when it doesn't matter if the property is a function or not.

Hashes and associative arrays

In some programming languages, there is a distinction between:

- A regular array, also called an **indexed** or **enumerated** array (the keys are numbers)
- An associative array, also called a **hash** or a dictionary (the keys are strings)

JavaScript uses arrays to represent indexed arrays and objects to represent associative arrays. If you want a hash in JavaScript, you use an object.

Accessing an object's properties

There are two ways to access a property of an object:

- Using the square bracket notation, for example `hero['occupation']`
- Using the dot notation, for example `hero.occupation`

The dot notation is easier to read and write, but it cannot always be used. The same rules apply as for quoting property names: if the name of the property is not a valid variable name, you cannot use the dot notation.

Let's take the `hero` object again:

```
var hero = {
  breed: 'Turtle',
  occupation: 'Ninja'
};
```

Accessing a property with the dot notation:

```
> hero.breed;
"Turtle"
```

Accessing a property with the bracket notation:

```
> hero['occupation'];
"Ninja"
```

Accessing a non-existing property returns `undefined`:

```
> 'Hair color is ' + hero.hair_color;
"Hair color is undefined"
```

Objects can contain any data, including other objects:

```
var book = {
  name: 'Catch-22',
  published: 1961,
  author: {
    firstname: 'Joseph',
    lastname: 'Heller'
  }
};
```

To get to the `firstname` property of the object contained in the `author` property of the `book` object, you use:

```
> book.author.firstname;
"Joseph"
```

Using the square brackets notation:

```
> book['author']['lastname'];
"Heller"
```

It works even if you mix both:

```
> book.author['lastname'];
"Heller"

> book['author'].lastname;
"Heller"
```

Another case where you need square brackets is when the name of the property you need to access is not known beforehand. During runtime, it's dynamically stored in a variable:

```
> var key = 'firstname';
> book.author[key];
"Joseph"
```

Calling an object's methods

You know a method is just a property that happens to be a function, so you access methods the same way as you would access properties: using the dot notation or using square brackets. Calling (invoking) a method is the same as calling any other function: you just add parentheses after the method name, which effectively says "Execute!".

```
> var hero = {
    breed: 'Turtle',
    occupation: 'Ninja',
    say: function () {
      return 'I am ' + hero.occupation;
    }
  };
> hero.say();
"I am Ninja"
```

If there are any parameters that you want to pass to a method, you proceed as with normal functions:

```
> hero.say('a', 'b', 'c');
```

Because you can use the array-like square brackets to access a property, this means you can also use brackets to access and invoke methods:

```
> hero['say']();
```

This is not a common practice unless the method name is not known at the time of writing code, but is instead defined at runtime:

```
var method = 'say';
hero[method]();
```

Best practice tip: no quotes (unless you have to)
Use the dot notation to access methods and properties and don't quote properties in your object literals.

Altering properties/methods

JavaScript allows you to alter the properties and methods of existing objects at any time. This includes adding new properties or deleting them. You can start with a "blank" object and add properties later. Let's see how you can go about doing this.

An object without properties is shown as follows:

```
> var hero = {};
```

A "blank" object

In this section, you started with a "blank" object, `var hero = {};`. Blank is in quotes because this object is not really empty and useless. Although at this stage it has no properties of its own, it has already inherited some. You'll learn more about own versus inherited properties later. So, an object in ES3 is never really "blank" or "empty". In ES5 though, there is a way to create a completely blank object that doesn't inherit anything, but let's not get ahead too much.

Accessing a non-existing property is shown as follows:

```
> typeof hero.breed;
"undefined"
```

Adding two properties and a method:

```
> hero.breed = 'turtle';
> hero.name = 'Leonardo';
> hero.sayName = function () {
    return hero.name;
  };
```

Calling the method:

```
> hero.sayName();
"Leonardo"
```

Deleting a property:

```
> delete hero.name;
true
```

Calling the method again will no longer find the deleted `name` property:

```
> hero.sayName();
"undefined"
```

Malleable objects

You can always change any object at any time, such as adding and removing properties and changing their values. But, there are exceptions to this rule. A few properties of some built-in objects are not changeable (for example, `Math.PI`, as you'll see later). Also, ES5 allows you to prevent changes to objects; you'll learn more about it in *Chapter 11, Built-in Objects*.

Using the this value

In the previous example, the `sayName()` method used `hero.name` to access the `name` property of the `hero` object. When you're inside a method though, there is another way to access the object the method belongs to: by using the special value `this`.

```
> var hero = {
    name: 'Rafaelo',
    sayName: function () {
      return this.name;
    }
  };
> hero.sayName();
"Rafaelo"
```

So, when you say `this`, you're actually saying "this object" or "the current object".

Constructor functions

There is another way to create objects: by using constructor functions. Let's see an example:

```
function Hero() {
  this.occupation = 'Ninja';
}
```

In order to create an object using this function, you use the `new` operator, like this:

```
> var hero = new Hero();
> hero.occupation;
"Ninja"
```

A benefit of using constructor functions is that they accept parameters, which can be used when creating new objects. Let's modify the constructor to accept one parameter and assign it to the `name` property:

```
function Hero(name) {
  this.name = name;
  this.occupation = 'Ninja';
  this.whoAreYou = function () {
    return "I'm " +
           this.name +
           " and I'm a " +
           this.occupation;
  };
}
```

Now you can create different objects using the same constructor:

```
> var h1 = new Hero('Michelangelo');
> var h2 = new Hero('Donatello');
> h1.whoAreYou();
```

"I'm Michelangelo and I'm a Ninja"

```
> h2.whoAreYou();
```

"I'm Donatello and I'm a Ninja"

Best practice

By convention, you should capitalize the first letter of your constructor functions so that you have a visual clue that this is not intended to be called as a regular function.

If you call a function that is designed to be a constructor but you omit the `new` operator, this is not an error, but it doesn't give you the expected result.

```
> var h = Hero('Leonardo');
> typeof h;
```

"undefined"

What happened here? There is no `new` operator, so a new object was *not* created. The function was called like any other function, so `h` contains the value that the function returns. The function does not return anything (there's no `return`), so it actually returns **undefined**, which gets assigned to `h`.

In this case, what does `this` refer to? It refers to the global object.

The global object

You have already learned a bit about global variables (and how you should avoid them). You also know that JavaScript programs run inside a host environment (the browser for example). Now that you know about objects, it's time for the whole truth: the host environment provides a global object and all global variables are accessible as properties of the global object.

If your host environment is the web browser, the global object is called `window`. Another way to access the global object (and this is also true in most other environments) is to use `this` outside a constructor function, for example in the global program code outside any function.

As an illustration, you can declare a global variable outside any function, such as:

```
> var a = 1;
```

Then, you can access this global variable in various ways:

- As a variable `a`
- As a property of the global object, for example `window['a']` or `window.a`
- As a property of the global object referred to as `this`:

  ```
  > var a = 1;
  > window.a;
  1

  > this.a;
  1
  ```

Let's go back to the case where you define a constructor function and call it without the `new` operator. In such cases, `this` refers to the global object and all the properties set to `this` become properties of `window`.

Declaring a constructor function and calling it without `new` returns **"undefined"**:

```
> function Hero(name) {
    this.name = name;
  }
> var h = Hero('Leonardo');
> typeof h;
"undefined"

> typeof h.name;
TypeError: Cannot read property 'name' of undefined
```

Because you had `this` inside `Hero`, a global variable (a property of the global object) called `name` was created:

```
> name;
"Leonardo"

> window.name;
"Leonardo"
```

If you call the same constructor function using `new`, then a new object is returned and `this` refers to it:

```
> var h2 = new Hero('Michelangelo');
> typeof h2;
"object"

> h2.name;
"Michelangelo"
```

The built-in global functions you have seen in *Chapter 3*, *Functions*, can also be invoked as methods of the `window` object. So, the following two calls have the same result:

```
> parseInt('101 dalmatians');
101

> window.parseInt('101 dalmatians');
101
```

And, when outside a function called as a constructor (with `new`), also:

```
> this.parseInt('101 dalmatians');
101
```

The constructor property

When an object is created, a special property is assigned to it behind the scenes—the `constructor` property. It contains a reference to the constructor function used to create this object.

Continuing from the previous example:

```
> h2.constructor;
function Hero(name) {
  this.name = name;
}
```

Because the `constructor` property contains a reference to a function, you might as well call this function to produce a new object. The following code is like saying, "I don't care how object `h2` was created, but I want another one just like it":

```
> var h3 = new h2.constructor('Rafaello');
> h3.name;
"Rafaello"
```

If an object was created using the object literal notation, its constructor is the built-in `Object()` constructor function (there is more about this later in this chapter):

```
> var o = {};
> o.constructor;
function Object() { [native code] }

> typeof o.constructor;
"function"
```

The instanceof operator

With the `instanceof` operator, you can test if an object was created with a specific constructor function:

```
> function Hero() {}
> var h = new Hero();
> var o = {};
> h instanceof Hero;
true

> h instanceof Object;
true

> o instanceof Object;
true
```

Note that you don't put parentheses after the function name (you don't use `h instanceof Hero()`). This is because you're not invoking this function, but just referring to it by name, as with any other variable.

Functions that return objects

In addition to using constructor functions and the `new` operator to create objects, you can also use a normal function to create objects without `new`. You can have a function that does a bit of preparatory work and has an object as a return value.

For example, here's a simple `factory()` function that produces objects:

```
function factory(name) {
  return {
    name: name
  };
}
```

Using the `factory()` function:

```
> var o = factory('one');
> o.name;
"one"

> o.constructor;
function Object() { [native code] }
```

In fact, you can also use constructor functions and return objects different from `this`. This means you can modify the default behavior of the constructor function. Let's see how.

Here's the normal constructor scenario:

```
> function C() {
    this.a = 1;
  }
> var c = new C();
> c.a;
1
```

But now look at this scenario:

```
> function C2() {
    this.a = 1;
    return {b: 2};
  }
> var c2 = new C2();
> typeof c2.a;
"undefined"

> c2.b;
2
```

What happened here? Instead of returning the object `this`, which contains the property `a`, the constructor returned another object that contains the property `b`. This is possible only if the return value is an object. Otherwise, if you try to return anything that is not an object, the constructor will proceed with its usual behavior and return `this`.

If you think about how objects are created inside constructor functions, you can imagine that a variable called `this` is defined at the top of the function and then returned at the end. It's as if something like this happens:

```
function C() {
  // var this = {}; // pseudo code, you can't do this
  this.a = 1;
  // return this;
}
```

Passing objects

When you assign an object to a different variable or pass it to a function, you only pass a reference to that object. Consequently, if you make a change to the reference, you're actually modifying the original object.

Here's an example of how you can assign an object to another variable and then make a change to the copy. As a result, the original object is also changed:

```
> var original = {howmany: 1};
> var mycopy = original;
> mycopy.howmany;
1

> mycopy.howmany = 100;
100

> original.howmany;
100
```

The same thing applies when passing objects to functions:

```
> var original = {howmany: 100};
> var nullify = function (o) { o.howmany = 0; };
> nullify(original);
> original.howmany;
0
```

Comparing objects

When you compare objects, you'll get `true` only if you compare two references to the same object. Comparing two distinct objects that happen to have the exact same methods and properties returns `false`.

Let's create two objects that look the same:

```
> var fido  = {breed: 'dog'};
> var benji = {breed: 'dog'};
```

Comparing them returns `false`:

```
> benji === fido;
false

> benji == fido;
false
```

You can create a new variable, `mydog`, and assign one of the objects to it. This way, `mydog` actually points to the same object:

```
> var mydog = benji;
```

In this case, `benji` is `mydog` because they are the same object (changing the `mydog` variable's properties will change the `benji` variable's properties). The comparison returns **true**:

```
> mydog === benji;
true
```

And, because `fido` is a different object, it does not compare to `mydog`:

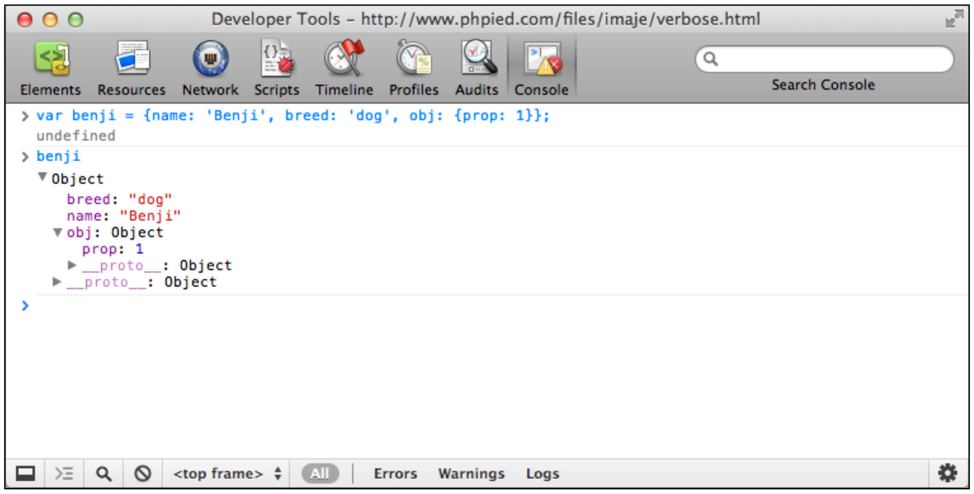
```
> mydog === fido;
false
```

Objects in the WebKit console

Before diving into the built-in objects in JavaScript, let's quickly say a few words about working with objects in the WebKit console.

After playing around with the examples in this chapter, you might have already noticed how objects are displayed in the console. If you create an object and type its name, you'll get an arrow pointing to the word **Object**.

The object is clickable and expands to show you a list of all of the properties of the object. If a property is also an object, there is an arrow next to it too, so you can expand this as well. This is handy as it gives you an insight into exactly what this object contains.

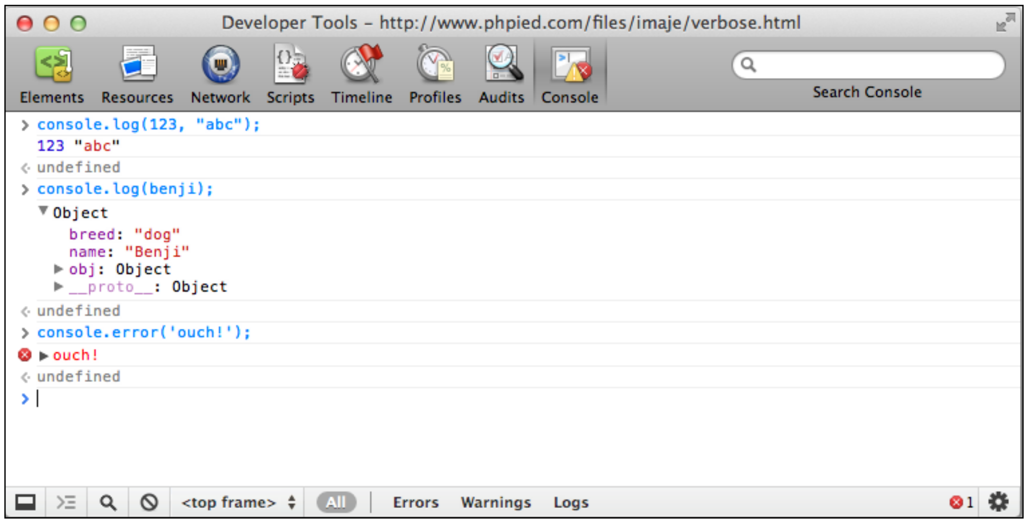


```
> var benji = {name: 'Benji', breed: 'dog', obj: {prop: 1}};
  undefined
> benji
  ▼Object
      breed: "dog"
      name: "Benji"
    ▼obj: Object
        prop: 1
      ▶__proto__: Object
    ▶__proto__: Object
>
```

You can ignore `__proto__` for now; there's more about it in the next chapter.

console.log

The console also offers you an object called `console` and a few methods, such as `console.log()` and `console.error()`, which you can use to display any value you want in the console.



```
> console.log(123, "abc");
  123 "abc"
< undefined
> console.log(benji);
  ▼Object
     breed: "dog"
     name: "Benji"
   ▶obj: Object
   ▶__proto__: Object
< undefined
> console.error('ouch!');
  ▶ouch!
< undefined
> |
```

`console.log()` is convenient when you want to quickly test something, as well as in your real scripts when you want to dump some intermediate debugging information. Here's how you can experiment with loops for example:

```
> for (var i = 0; i < 5; i++) {
    console.log(i);
  }
0
1
2
3
4
```

Built-in objects

Earlier in this chapter, you came across the `Object()` constructor function. It's returned when you create objects with the object literal notation and access their `constructor` property. `Object()` is one of the built-in constructors; there are a few others, and in the rest of this chapter you'll see all of them.

The built-in objects can be divided into three groups:

- **Data wrapper objects**: These are `Object`, `Array`, `Function`, `Boolean`, `Number`, and `String`. These objects correspond to the different data types in JavaScript. There is a data wrapper object for every different value returned by `typeof` (discussed in *Chapter 2*, *Primitive Data Types, Arrays, Loops, and Conditions*), with the exception of "undefined" and "null".
- **Utility objects**: These are `Math`, `Date`, and `RegExp`, and can come in handy.
- **Error objects**: These include the generic `Error` object as well as other more specific objects that can help your program recover its working state when something unexpected happens.

Only a handful of methods of the built-in objects will be discussed in this chapter. For a full reference, see *Chapter 11*, *Built-in Objects*.

If you're confused about what a built-in object is and what a built-in constructor is, well, they are the same thing. In a moment, you'll see how functions, and therefore constructor functions, are also objects.

Object

`Object` is the parent of all JavaScript objects, which means that every object you create inherits from it. To create a new "empty" object, you can use the literal notation or the `Object()` constructor function. The following two lines are equivalent:

```
> var o = {};
> var o = new Object();
```

As mentioned before, an "empty" (or "blank") object is not completely useless because it already contains several *inherited* methods and properties. In this book, "empty" means an object like `{}` that has no properties of its own other than the ones it automatically gets. Let's see a few of the properties that even "blank" objects already have:

- The `o.constructor` property returns a reference to the constructor function
- `o.toString()` is a method that returns a string representation of the object
- `o.valueOf()` returns a single-value representation of the object; often this is the object itself

Let's see these methods in action. First, create an object:

```
> var o = new Object();
```

Calling `toString()` returns a string representation of the object:

```
> o.toString();
"[object Object]"
```

`toString()` will be called internally by JavaScript when an object is used in a string context. For example, `alert()` works only with strings, so if you call the `alert()` function passing an object, the `toString()`method will be called behind the scenes. These two lines produce the same result:

```
> alert(o);
> alert(o.toString());
```

Another type of string context is the string concatenation. If you try to concatenate an object with a string, the object's `toString()` method is called first:

```
> "An object: " + o;
"An object: [object Object]"
```

`valueOf()` is another method that all objects provide. For the simple objects (whose constructor is `Object()`), the `valueOf()` method returns the object itself:

```
> o.valueOf() === o;
true
```

To summarize:

- You can create objects either with `var o = {};` (object literal notation, the preferred method) or with `var o = new Object();`
- Any object, no matter how complex, inherits from the `Object` object, and therefore offers methods such as `toString()` and properties such as `constructor`

Array

`Array()` is a built-in function that you can use as a constructor to create arrays:

```
> var a = new Array();
```

This is equivalent to the array literal notation:

```
> var a = [];
```

No matter how the array is created, you can add elements to it as usual:

```
> a[0] = 1;
> a[1] = 2;
> a;
[1, 2]
```

When using the `Array()` constructor, you can also pass values that will be assigned to the new array's elements:

```
> var a = new Array(1, 2, 3, 'four');
> a;
[1, 2, 3, "four"]
```

An exception to this is when you pass a single number to the constructor. In this case, the number is considered to be the length of the array:

```
> var a2 = new Array(5);
> a2;
[undefined x 5]
```

Because arrays are created with a constructor, does this mean that arrays are in fact objects? Yes, and you can verify this by using the `typeof` operator:

```
> typeof [1, 2, 3];
"object"
```

Because arrays are objects, this means that they inherit the properties and methods of the parent `Object`:

```
> var a = [1, 2, 3, 'four'];
> a.toString();
"1,2,3,four"

> a.valueOf();
[1, 2, 3, "four"]

> a.constructor;
function Array() { [native code] }
```

Arrays are objects, but of a special type because:

- The names of their properties are automatically assigned using numbers starting from 0
- They have a `length` property that contains the number of elements in the array
- They have more built-in methods in addition to those inherited from the parent `Object`

Let's examine the differences between an array and an object, starting by creating the empty array `a` and the empty object `o`:

```
> var a = [], o = {};
```

Array objects have a `length` property automatically defined for them, while normal objects do not:

```
> a.length;
0

> typeof o.length;
"undefined"
```

It's OK to add both numeric and non-numeric properties to both arrays and objects:

```
> a[0] = 1;
> o[0] = 1;
> a.prop = 2;
> o.prop = 2;
```

The `length` property is always up-to-date with the number of numeric properties, while it ignores the non-numeric ones:

```
> a.length;
1
```

The `length` property can also be set by you. Setting it to a greater value than the current number of items in the array makes room for additional elements. If you try to access these non-existing elements, you'll get the value `undefined`:

```
> a.length = 5;
5

> a;
[1, undefined x 4]
```

Setting the `length` property to a lower value removes the trailing elements:

```
> a.length = 2;
2

> a;
[1, undefined x 1]
```

A few array methods

In addition to the methods inherited from the parent `Object`, array objects also have specialized methods for working with arrays, such as `sort()`, `join()`, and `slice()`, among others (see *Chapter 11*, *Built-in Objects*, for the full list).

Let's take an array and experiment with some of these methods:

```
> var a = [3, 5, 1, 7, 'test'];
```

The `push()` method appends a new element to the end of the array. The `pop()` method removes the last element. `a.push('new')` works like `a[a.length] = 'new'` and `a.pop()` is like `a.length--`.

`push()` returns the length of the changed array, whereas `pop()` returns the removed element:

```
> a.push('new');
6

> a;
 [3, 5, 1, 7, "test", "new"]

> a.pop();
"new"

> a;
 [3, 5, 1, 7, "test"]
```

The `sort()` method sorts the array and returns it. In the next example, after the sort, both `a` and `b` point to the same array:

```
> var b = a.sort();
> b;
[1, 3, 5, 7, "test"]

> a === b;
true
```

The `join()` method returns a string containing the values of all the elements in the array glued together using the string parameter passed to `join()`:

```
> a.join(' is not ');
"1 is not 3 is not 5 is not 7 is not test"
```

The `slice()` method returns a piece of the array without modifying the source array. The first parameter to `slice()` is the start index (zero-based) and the second is the end index (both indices are zero-based):

```
> b = a.slice(1, 3);
[3, 5]

> b = a.slice(0, 1);
[1]

> b = a.slice(0, 2);
[1, 3]
```

After all the slicing, the source array is still the same:

```
> a;
[1, 3, 5, 7, "test"]
```

The `splice()` method modifies the source array. It removes a slice, returns it, and optionally fills the gap with new elements. The first two parameters define the start index and length (number of elements) of the slice to be removed; the other parameters pass the new values:

```
> b = a.splice(1, 2, 100, 101, 102);
[3, 5]

> a;
[1, 100, 101, 102, 7, "test"]
```

Filling the gap with new elements is optional and you can skip it:

```
> a.splice(1, 3);
[100, 101, 102]

> a;
[1, 7, "test"]
```

Function

You already know that functions are a special data type. But, it turns out that there's more to it than that: functions are actually objects. There is a built-in constructor function called `Function()` that allows for an alternative (but not necessarily recommended) way to create a function.

The following example shows three ways to define a function:

```
> function sum(a, b) { // function declaration
    return a + b;
  }
```

```
> sum(1, 2);
3

> var sum = function (a, b) { // function expression
    return a + b;
  };
> sum(1, 2)
3

> var sum = new Function('a', 'b', 'return a + b;');
> sum(1, 2)
3
```

When using the `Function()` constructor, you pass the parameter names first (as strings) and then the source code for the body of the function (again as a string). The JavaScript engine needs to evaluate the source code you pass and create the new function for you. This source code evaluation suffers from the same drawbacks as the `eval()` function, so defining functions using the `Function()` constructor should be avoided when possible.

If you use the `Function()` constructor to create functions that have lots of parameters, bear in mind that the parameters can be passed as a single comma-delimited list; so, for example, these are the same:

```
> var first = new Function(
    'a, b, c, d',
    'return arguments;'
  );
> first(1, 2, 3, 4);
   [1, 2, 3, 4]

> var second = new Function(
    'a, b, c',
    'd',
    'return arguments;'
  );
> second(1, 2, 3, 4);
   [1, 2, 3, 4]

> var third = new Function(
    'a',
    'b',
    'c',
    'd',
    'return arguments;'
  );
> third(1, 2, 3, 4);
   [1, 2, 3, 4]
```

Best practice

Do not use the `Function()` constructor. As with `eval()` and `setTimeout()` (discussed later in the book), always try to stay away from passing JavaScript code as a string.

Properties of function objects

Like any other object, functions have a `constructor` property that contains a reference to the `Function()` constructor function. This is true no matter which syntax you used to create the function.

```
> function myfunc(a) {
    return a;
  }
> myfunc.constructor;
function Function() { [native code] }
```

Functions also have a `length` property, which contains the number of formal parameters the function expects.

```
> function myfunc(a, b, c) {
    return true;
  }
> myfunc.length;
   3
```

Prototype

One of the most widely used properties of function objects is the `prototype` property. You'll see this property discussed in detail in the next chapter, but for now, let's just say:

- The `prototype` property of a function object points to another object
- Its benefits shine only when you use this function as a constructor
- All objects created with this function keep a reference to the `prototype` property and can use its properties as their own

Let's see a quick example to demonstrate the `prototype` property. Take a simple object that has a property `name` and a method `say()`.

```
var ninja = {
  name: 'Ninja',
  say: function () {
    return 'I am a ' + this.name;
  }
};
```

When you create a function (even one without a body), you can verify that it automatically has a `prototype` property that points to a new object.

```
> function F() {}
> typeof F.prototype;
"object"
```

It gets interesting when you modify the `prototype` property. You can add properties to it or you can replace the default object with any other object. Let's assign `ninja` to the prototype.

```
> F.prototype = ninja;
```

Now, and here's where the magic happens, using the function `F()` as a constructor function, you can create a new object, `baby_ninja`, which will have access to the properties of `F.prototype` (which points to `ninja`) as if it were its own.

```
> var baby_ninja = new F();
> baby_ninja.name;
"Ninja"

> baby_ninja.say();
"I am a Ninja"
```

There will be much more on this topic later. In fact, the whole next chapter is about the `prototype` property.

Methods of function objects

Function objects, being a descendant of the top parent `Object`, get the default methods such as `toString()`. When invoked on a function, the `toString()` method returns the source code of the function.

```
> function myfunc(a, b, c) {
    return a + b + c;
  }
> myfunc.toString();
"function myfunc(a, b, c) {
 return a + b + c;
}"
```

If you try to peek into the source code of the built-in functions, you'll get the string `[native code]` instead of the body of the function.

```
> parseInt.toString();
"function parseInt() { [native code] }"
```

As you can see, you can use `toString()` to differentiate between native methods and developer-defined ones.

> The behavior of the function's `toString()` is environment-dependent, and it does differ among browsers in terms of spacing and new lines.

Call and apply

Function objects have `call()` and `apply()` methods. You can use them to invoke a function and pass any arguments to it.

These methods also allow your objects to "borrow" methods from other objects and invoke them as their own. This is an easy and powerful way to reuse code.

Let's say you have a `some_obj` object, which contains the method `say()`.

```
var some_obj = {
  name: 'Ninja',
  say: function (who) {
    return 'Haya ' + who + ', I am a ' + this.name;
  }
};
```

You can call the `say()` method, which internally uses `this.name` to gain access to its own `name` property.

```
> some_obj.say('Dude');
"Haya Dude, I am a Ninja"
```

Now let's create a simple object, `my_obj`, which only has a `name` property.

```
> var my_obj = {name: 'Scripting guru'};
```

`my_obj` likes the `some_obj` object's `say()` method so much that it wants to invoke it as its own. This is possible using the `call()` method of the `say()` function object.

```
> some_obj.say.call(my_obj, 'Dude');
"Haya Dude, I am a Scripting guru"
```

It worked! But what happened here? You invoked the `call()` method of the `say()` function object passing two parameters: the object `my_obj` and the string `'Dude'`. The result is that when `say()` is invoked, the references to the `this` value that it contains point to `my_obj`. This way, `this.name` doesn't return **Ninja**, but **Scripting guru** instead.

If you have more parameters to pass when invoking the `call()` method, you just keep adding them.

```
some_obj.someMethod.call(my_obj, 'a', 'b', 'c');
```

If you don't pass an object as a first parameter to `call()` or you pass `null`, the global object is assumed.

The method `apply()` works the same way as `call()`, but with the difference that all parameters you want to pass to the method of the other object are passed as an array. The following two lines are equivalent:

```
some_obj.someMethod.apply(my_obj, ['a', 'b', 'c']);
some_obj.someMethod.call(my_obj, 'a', 'b', 'c');
```

Continuing the previous example, you can use:

```
> some_obj.say.apply(my_obj, ['Dude']);
"Haya Dude, I am a Scripting guru"
```

The arguments object revisited

In the previous chapter, you have seen how, from inside a function, you have access to something called `arguments`, which contains the values of all the parameters passed to the function:

```
> function f() {
    return arguments;
  }
> f(1, 2, 3);
[1, 2, 3]
```

`arguments` looks like an array, but it is actually an array-like object. It resembles an array because it contains indexed elements and a `length` property. However, the similarity ends there, as `arguments` doesn't provide any of the array methods, such as `sort()` or `slice()`.

However, you can convert `arguments` to an array and benefit from all the array goodies. Here's what you can do, practicing your newly-learned `call()` method:

```
> function f() {
    var args = [].slice.call(arguments);
    return args.reverse();
  }

> f(1, 2, 3, 4);
[4, 3, 2, 1]
```

As you can see, you can borrow `slice()` using `[].slice` or the more verbose `Array.prototype.slice`.

Inferring object types

You can see that you have this array-like `arguments` object looking so much like an array object. How can you reliably tell the difference between the two? Additionally, `typeof` returns `object` when used with arrays. Therefore, how can you tell the difference between an object and an array?

The silver bullet is the `Object` object's `toString()` method. It gives you the internal class name used to create a given object.

```
> Object.prototype.toString.call({});
"[object Object]"

> Object.prototype.toString.call([]);
"[object Array]"
```

You have to call the original `toString()` method as defined in the prototype of the `Object` constructor. Otherwise, if you call the `Array` function's `toString()`, it will give you a different result, as it's been overridden for the specific purposes of the array objects:

```
> [1, 2, 3].toString();
"1,2,3"
```

This is the same as:

```
> Array.prototype.toString.call([1, 2, 3]);
"1,2,3"
```

Let's have some more fun with `toString()`. Make a handy reference to save typing:

```
> var toStr = Object.prototype.toString;
```

Differentiate between an array and the array-like object `arguments`:

```
> (function () {
    return toStr.call(arguments);
  }());
"[object Arguments]"
```

You can even inspect DOM elements:

```
> toStr.call(document.body);
"[object HTMLBodyElement]"
```

Boolean

Your journey through the built-in objects in JavaScript continues, and the next three are fairly straightforward; they merely wrap the primitive data types Boolean, number, and string.

You already know a lot about Booleans from *Chapter 2*, *Primitive Data Types, Arrays, Loops, and Conditions*. Now, let's meet the `Boolean()` constructor:

```
> var b = new Boolean();
```

It's important to note that this creates a new object, `b`, and not a primitive Boolean value. To get the primitive value, you can call the `valueOf()` method (inherited from `Object` and customized):

```
> var b = new Boolean();
> typeof b;
"object"

> typeof b.valueOf();
"boolean"

> b.valueOf();
false
```

Overall, objects created with the `Boolean()` constructor are not too useful, as they don't provide any methods or properties other than the inherited ones.

The `Boolean()` function, when called as a normal function without `new`, converts non-Booleans to Booleans (which is like using a double negation `!!value`):

```
> Boolean("test");
true

> Boolean("");
false

> Boolean({});
true
```

Apart from the six falsy values, everything else is true in JavaScript, including all objects. This also means that all Boolean objects created with `new Boolean()` are also true, as they are objects:

```
> Boolean(new Boolean(false));
true
```

This can be confusing, and since Boolean objects don't offer any special methods, it's best to just stick with regular primitive Boolean values.

Number

Similarly to `Boolean()`, the `Number()` function can be used as:

- A constructor function (with `new`) to create objects.
- A normal function in order to try to convert any value to a number. This is similar to the use of `parseInt()` or `parseFloat()`.

```
> var n = Number('12.12');
> n;
12.12

> typeof n;
"number"

> var n = new Number('12.12');
> typeof n;
"object"
```

Because functions are objects, they can also have properties. The `Number()` function has constant built-in properties that you cannot modify:

```
> Number.MAX_VALUE;
1.7976931348623157e+308

> Number.MIN_VALUE;
5e-324

> Number.POSITIVE_INFINITY;
Infinity

> Number.NEGATIVE_INFINITY;
-Infinity

> Number.NaN;
NaN
```

The number objects provide three methods: `toFixed()`, `toPrecision()`, and `toExponential()` (see *Chapter 11, Built-in Objects*, for more details):

```
> var n = new Number(123.456);
> n.toFixed(1);
"123.5"
```

Note that you can use these methods without explicitly creating a number object first. In such cases, the number object is created (and destroyed) for you behind the scenes:

```
> (12345).toExponential();
"1.2345e+4"
```

Like all objects, number objects also provide the `toString()` method. When used with number objects, this method accepts an optional radix parameter (10 being the default):

```
> var n = new Number(255);
> n.toString();
"255"

> n.toString(10);
"255"

> n.toString(16);
"ff"

> (3).toString(2);
"11"

> (3).toString(10);
"3"
```

String

You can use the `String()` constructor function to create string objects. String objects provide convenient methods for text manipulation.

Here's an example that shows the difference between a string object and a primitive string data type:

```
> var primitive = 'Hello';
> typeof primitive;
"string"

> var obj = new String('world');
> typeof obj;
"object"
```

A string object is similar to an array of characters. String objects have an indexed property for each character (introduced in ES5, but long supported in many browsers except old IEs) and they also have a `length` property.

```
> obj[0];
"w"

> obj[4];
"d"

> obj.length;
5
```

To extract the primitive value from the string object, you can use the `valueOf()` or `toString()` methods inherited from `Object`. You'll probably never need to do this, as `toString()` is called behind the scenes if you use an object in a primitive string context.

```
> obj.valueOf();
"world"

> obj.toString();
"world"

> obj + "";
"world"
```

Primitive strings are not objects, so they don't have any methods or properties. But, JavaScript also offers you the syntax to treat primitive strings as objects (just like you saw already with primitive numbers).

In the following example, string objects are being created (and then destroyed) behind the scenes every time you treat a primitive string as if it were an object:

```
> "potato".length;
6

> "tomato"[0];
"t"

> "potatoes"["potatoes".length - 1];
"s"
```

One final example to illustrate the difference between a string primitive and a string object: let's convert them to Boolean. The empty string is a falsy value, but any string object is truthy (because all objects are truthy):

```
> Boolean("");
false

> Boolean(new String(""));
true
```

Similarly to `Number()` and `Boolean()`, if you use the `String()` function without `new`, it converts the parameter to a primitive:

```
> String(1);
"1"
```

If you pass an object to `String()`, this object's `toString()` method will be called first:

```
> String({p: 1});
    "[object Object]"

> String([1, 2, 3]);
    "1,2,3"

> String([1, 2, 3]) === [1, 2, 3].toString();
    true
```

A few methods of string objects

Let's experiment with a few of the methods you can call on string objects (see *Chapter 11*, *Built-in Objects*, for the full list).

Start off by creating a string object:

```
> var s = new String("Couch potato");
```

`toUpperCase()` and `toLowerCase()` transforms the capitalization of the string:

```
> s.toUpperCase();
"COUCH POTATO"

> s.toLowerCase();
"couch potato"
```

`charAt()` tells you the character found at the position you specify, which is the same as using square brackets (treating a string as an array of characters):

```
> s.charAt(0);
"C"

> s[0];
"C"
```

If you pass a non-existing position to `charAt()`, you get an empty string:

```
> s.charAt(101);
""
```

`indexOf()` allows you to search within a string. If there is a match, the method returns the position at which the first match is found. The position count starts at 0, so the second character in "Couch" is "o" at position 1:

```
> s.indexOf('o');
1
```

You can optionally specify where (at what position) to start the search. The following finds the second "o", because `indexOf()` is instructed to start the search at position 2:

```
> s.indexOf('o', 2);
7
```

`lastIndexOf()` starts the search from the end of the string (but the position of the match is still counted from the beginning):

```
> s.lastIndexOf('o');
11
```

You can also search for strings, not only characters, and the search is case sensitive:

```
> s.indexOf('Couch');
0
```

If there is no match, the function returns position -1:

```
> s.indexOf('couch');
-1
```

For a case-insensitive search, you can transform the string to lowercase first and then search:

```
> s.toLowerCase().indexOf('couch');
0
```

When you get 0, this means that the matching part of the string starts at position 0. This can cause confusion when you check with `if`, because `if` converts the position 0 to a Boolean `false`. So, while this is syntactically correct, it is logically wrong:

```
if (s.indexOf('Couch')) {...}
```

The proper way to check if a string contains another string is to compare the result of `indexOf()` to the number `-1`:

```
if (s.indexOf('Couch') !== -1) {...}
```

`slice()` and `substring()` return a piece of the string when you specify start and end positions:

```
> s.slice(1, 5);
"ouch"

> s.substring(1, 5);
"ouch"
```

Note that the second parameter you pass is the end position, not the length of the piece. The difference between these two methods is how they treat negative arguments. `substring()` treats them as zeros, while `slice()` adds them to the length of the string. So, if you pass parameters `(1, -1)` to both methods, it's the same as `substring(1, 0)` and `slice(1, s.length - 1)`:

```
> s.slice(1, -1);
"ouch potat"

> s.substring(1, -1);
"C"
```

There's also the non-standard method `substr()`, but you should try to avoid it in favor of `substring()`.

The `split()` method creates an array from the string using another string that you pass as a separator:

```
> s.split(" ");
 ["Couch", "potato"]
```

`split()` is the opposite of `join()`, which creates a string from an array:

```
> s.split(' ').join(' ');
"Couch potato"
```

`concat()` glues strings together, the way the `+` operator does for primitive strings:

```
> s.concat("es");
"Couch potatoes"
```

Note that while some of the preceding methods discussed return new primitive strings, none of them modify the source string. After all the method calls listed previously, the initial string is still the same:

```
> s.valueOf();
"Couch potato"
```

You have seen how to use `indexOf()` and `lastIndexOf()` to search within strings, but there are more powerful methods (`search()`, `match()`, and `replace()`) that take regular expressions as parameters. You'll see these later in the `RegExp()` constructor function.

At this point, you're done with all of the data wrapper objects, so let's move on to the utility objects `Math`, `Date`, and `RegExp`.

Math

`Math` is a little different from the other built-in global objects you have seen previously. It's not a function, and therefore cannot be used with `new` to create objects. `Math` is a built-in global object that provides a number of methods and properties for mathematical operations.

The `Math` object's properties are constants, so you can't change their values. Their names are all in uppercase to emphasize the difference between them and a normal property (similar to the constant properties of the `Number()` constructor). Let's see a few of these constant properties:

- The constant π:
  ```
  > Math.PI;
    3.141592653589793
  ```
- Square root of 2:
  ```
  > Math.SQRT2;
    1.4142135623730951
  ```
- Euler's constant:
  ```
  > Math.E;
    2.718281828459045
  ```
- Natural logarithm of 2:
  ```
  > Math.LN2;
    0.6931471805599453
  ```

- Natural logarithm of 10:

```
> Math.LN10;
  2.302585092994046
```

Now you know how to impress your friends the next time they (for whatever reason) start wondering, "What was the value of *e*? I can't remember." Just type `Math.E` in the console and you have the answer.

Let's take a look at some of the methods the `Math` object provides (the full list is in *Chapter 11*, *Built-in Objects*).

Generating random numbers:

```
> Math.random();
0.3649461670235814
```

The `random()` function returns a number between 0 and 1, so if you want a number between, let's say, 0 and 100, you can do the following:

```
> 100 * Math.random();
```

For numbers between any two values, use the formula `((max - min) * Math.random()) + min`. For example, a random number between 2 and 10 would be:

```
> 8 * Math.random() + 2;
9.175650496668485
```

If you only need an integer, you can use one of the following rounding methods:

- `floor()` to round down
- `ceil()` to round up
- `round()` to round to the nearest

For example, to get either 0 or 1:

```
> Math.round(Math.random());
```

If you need the lowest or the highest among a set of numbers, you have the `min()` and `max()` methods. So, if you have a form on a page that asks for a valid month, you can make sure that you always work with sane data (a value between 1 and 12):

```
> Math.min(Math.max(1, input), 12);
```

The `Math` object also provides the ability to perform mathematical operations for which you don't have a designated operator. This means that you can raise to a power using `pow()`, find the square root using `sqrt()`, and perform all the trigonometric operations—`sin()`, `cos()`, `atan()`, and so on.

For example, to calculate 2 to the power of 8:

```
> Math.pow(2, 8);
256
```

And to calculate the square root of 9:

```
> Math.sqrt(9);
3
```

Date

`Date()` is a constructor function that creates date objects. You can create a new object by passing:

- Nothing (defaults to today's date)
- A date-like string
- Separate values for day, month, time, and so on
- A timestamp

Following is an object instantiated with today's date/time:

```
> new Date();
Wed Feb 27 2013 23:49:28 GMT-0800 (PST)
```

The console displays the result of the `toString()` method called on the date object, so you get this long string **Wed Feb 27 2013 23:49:28 GMT-0800 (PST)** as a representation of the date object.

Here are a few examples of using strings to initialize a date object. Note how many different formats you can use to specify the date:

```
> new Date('2015 11 12');
Thu Nov 12 2015 00:00:00 GMT-0800 (PST)

> new Date('1 1 2016');
Fri Jan 01 2016 00:00:00 GMT-0800 (PST)

> new Date('1 mar 2016 5:30');
Tue Mar 01 2016 05:30:00 GMT-0800 (PST)
```

The `Date` constructor can figure out a date from different strings, but this is not really a reliable way of defining a precise date, for example when passing user input to the constructor. The better way is to pass numeric values to the `Date()` constructor representing:

- Year

- Month: 0 (January) to 11 (December)
- Day: 1 to 31
- Hour: 0 to 23
- Minutes: 0 to 59
- Seconds: 0 to 59
- Milliseconds: 0 to 999

Let's see some examples.

Passing all the parameters:

```
> new Date(2015, 0, 1, 17, 05, 03, 120);
Tue Jan 01 2015 17:05:03 GMT-0800 (PST)
```

Passing date and hour:

```
> new Date(2015, 0, 1, 17);
Tue Jan 01 2015 17:00:00 GMT-0800 (PST)
```

Watch out for the fact that the month starts from 0, so 1 is February:

```
> new Date(2016, 1, 28);
Sun Feb 28 2016 00:00:00 GMT-0800 (PST)
```

If you pass a greater than allowed value, your date "overflows" forward. Because there's no February 30 in 2016, this means it has to be March 1st (2016 is a leap year):

```
> new Date(2016, 1, 29);
Mon Feb 29 2016 00:00:00 GMT-0800 (PST)
```

```
> new Date(2016, 1, 30);
Tue Mar 01 2016 00:00:00 GMT-0800 (PST)
```

Similarly, December 32nd becomes January 1st of the next year:

```
> new Date(2012, 11, 31);
Mon Dec 31 2012 00:00:00 GMT-0800 (PST)
```

```
> new Date(2012, 11, 32);
Tue Jan 01 2013 00:00:00 GMT-0800 (PST)
```

Finally, a date object can be initialized with a timestamp (the number of milliseconds since the UNIX epoch, where 0 milliseconds is January 1, 1970):

```
> new Date(1357027200000);
Tue Jan 01 2013 00:00:00 GMT-0800 (PST)
```

If you call `Date()` without `new`, you get a string representing the current date, whether or not you pass any parameters. The following example gives the current time (current when this example was run):

```
> Date();
Wed Feb 27 2013 23:51:46 GMT-0800 (PST)

> Date(1, 2, 3, "it doesn't matter");
Wed Feb 27 2013 23:51:52 GMT-0800 (PST)

> typeof Date();
"string"

> typeof new Date();
"object"
```

Methods to work with date objects

Once you've created a date object, there are lots of methods you can call on that object. Most of the methods can be divided into `set*()` and `get*()` methods, for example, `getMonth()`, `setMonth()`, `getHours()`, `setHours()`, and so on. Let's see some examples.

Creating a date object:

```
> var d = new Date(2015, 1, 1);
> d.toString();
Sun Feb 01 2015 00:00:00 GMT-0800 (PST)
```

Setting the month to March (months start from 0):

```
> d.setMonth(2);
1425196800000

> d.toString();
Sun Mar 01 2015 00:00:00 GMT-0800 (PST)
```

Getting the month:

```
> d.getMonth();
2
```

In addition to all the methods of date objects, there are also two methods (plus one more added in ES5) that are properties of the `Date()` function/object. These do not need a date object; they work just like the `Math` object's methods. In class-based languages, such methods would be called static because they don't require an instance.

`Date.parse()` takes a string and returns a timestamp:

```
> Date.parse('Jan 11, 2018');
1515657600000
```

`Date.UTC()` takes all the parameters for year, month, day, and so on, and produces a timestamp in Universal Time:

```
> Date.UTC(2018, 0, 11);
1515628800000
```

Because the `new Date()` constructor can accept timestamps, you can pass the result of `Date.UTC()` to it. Using the following example, you can see how `UTC()` works with Universal Time, while `new Date()` works with local time:

```
> new Date(Date.UTC(2018, 0, 11));
Wed Jan 10 2018 16:00:00 GMT-0800 (PST)

> new Date(2018, 0, 11);
Thu Jan 11 2018 00:00:00 GMT-0800 (PST)
```

The ES5 addition to the `Date` constructor is the method `now()`, which returns the current timestamp. It provides a more convenient way to get the timestamp instead of using the `getTime()` method on a date object as you would in ES3:

```
> Date.now();
1362038353044

> Date.now() === new Date().getTime();
true
```

You can think of the internal representation of the date being an integer timestamp and all other methods being "sugar" on top of it. So, it makes sense that the `valueOf()` is a timestamp:

```
> new Date().valueOf();
1362418306432
```

Also dates cast to integers with the + operator:

```
> +new Date();
1362418318311
```

Calculating birthdays

Let's see one final example of working with `Date` objects. I was curious about which day my birthday falls on in 2016:

```
> var d = new Date(2016, 5, 20);
> d.getDay();
1
```

Starting the count from 0 (Sunday), 1 means Monday. Is that so?

```
> d.toDateString();
"Mon Jun 20 2016"
```

OK, good to know, but Monday is not necessarily the best day for a party. So, how about a loop that shows how many times June 20th is a Friday from year 2016 to year 3016, or better yet, let's see the distribution of all the days of the week. After all, with all the progress in DNA hacking, we're all going to be alive and kicking in 3016.

First, let's initialize an array with seven elements, one for each day of the week. These will be used as counters. Then, as a loop goes up to 3016, let's increment the counters:

```
var stats = [0, 0, 0, 0, 0, 0, 0];
```

The loop:

```
for (var i = 2016; i < 3016; i++) {
   stats[new Date(i, 5, 20).getDay()]++;
}
```

And the result:

```
> stats;
[140, 146, 140, 145, 142, 142, 145]
```

142 Fridays and 145 Saturdays. Woo-hoo!

RegExp

Regular expressions provide a powerful way to search and manipulate text. Different languages have different implementations (think "dialects") of the regular expressions syntax. JavaScript uses the Perl 5 syntax.

Instead of saying "regular expression", people often shorten it to "regex" or "regexp".

A regular expression consists of:

- A pattern you use to match text
- Zero or more modifiers (also called flags) that provide more instructions on how the pattern should be used

The pattern can be as simple as literal text to be matched verbatim, but that's rare, and in such cases you're better off using `indexOf()`. Most of the time, the pattern is more complex and could be difficult to understand. Mastering regular expression's patterns is a large topic, which won't be discussed in full detail here; instead, you'll see what JavaScript provides in terms of syntax, objects, and methods in order to support the use of regular expressions. You can also refer to *Chapter 12*, *Regular Expressions*, to help you when you're writing patterns.

JavaScript provides the `RegExp()` constructor, which allows you to create regular expression objects:

```
> var re = new RegExp("j.*t");
```

There is also the more convenient *regexp literal notation*:

```
> var re = /j.*t/;
```

In the preceding example, `j.*t` is the regular expression pattern. It means "match any string that starts with `j`, ends with `t`, and has zero or more characters in between". The asterisk (`*`) means "zero or more of the preceding"; the dot (`.`) means "any character". The pattern needs to be quoted when passed to a `RegExp()` constructor.

Properties of RegExp objects

Regular expression objects have the following properties:

- `global`: If this property is `false`, which is the default, the search stops when the first match is found. Set this to `true` if you want all matches.
- `ignoreCase`: When the match is case insensitive, the defaults to `false` (meaning the default is a case sensitive match).
- `multiline`: Search matches that may span over more than one line default to `false`.
- `lastIndex`: The position at which to start the search; this defaults to 0.
- `source`: Contains the regexp pattern.

None of these properties, except for `lastIndex`, can be changed once the object has been created.

The first three items in the preceding list represent the regex modifiers. If you create a regex object using the constructor, you can pass any combination of the following characters as a second parameter:

- `g` for `global`
- `i` for `ignoreCase`
- `m` for `multiline`

These letters can be in any order. If a letter is passed, the corresponding modifier property is set to `true`. In the following example, all modifiers are set to `true`:

```
> var re = new RegExp('j.*t', 'gmi');
```

Let's verify:

```
> re.global;
true
```

Once set, the modifier cannot be changed:

```
> re.global = false;
> re.global;
true
```

To set any modifiers using the *regex literal*, you add them after the closing slash:

```
> var re = /j.*t/ig;
> re.global;
true
```

Methods of RegExp objects

Regex objects provide two methods you can use to find matches: `test()` and `exec()`. They both accept a string parameter. `test()` returns a Boolean (`true` when there's a match, `false` otherwise), while `exec()` returns an array of matched strings. Obviously, `exec()` is doing more work, so use `test()` unless you really need to do something with the matches. People often use regular expressions to validate data, in this case, `test()` should be enough.

In the following example, there is no match because of the capital `J`:

```
> /j.*t/.test("Javascript");
false
```

A case insensitive test gives a positive result:

```
> /j.*t/i.test("Javascript");
true
```

The same test using `exec()` returns an array, and you can access the first element as shown below:

```
> /j.*t/i.exec("Javascript")[0];
"Javascript"
```

String methods that accept regular expressions as arguments

Previously in this chapter, you learned about string objects and how you can use the `indexOf()` and `lastIndexOf()`methods to search within text. Using these methods, you can only specify literal string patterns to search. A more powerful solution would be to use regular expressions to find text. String objects offer you this ability.

String objects provide the following methods that accept regular expression objects as parameters:

- `match()` returns an array of matches
- `search()` returns the position of the first match
- `replace()` allows you to substitute matched text with another string
- `split()` also accepts a regexp when splitting a string into array elements

search() and match()

Let's see some examples of using the `search()` and `match()` methods. First, you create a string object:

```
> var s = new String('HelloJavaScriptWorld');
```

Using `match()`, you get an array containing only the first match:

```
> s.match(/a/);
["a"]
```

Using the `g` modifier, you perform a global search, so the result array contains two elements:

```
> s.match(/a/g);
["a", "a"]
```

A case insensitive match is as follows:

```
> s.match(/j.*a/i);
["Java"]
```

The `search()` method gives you the position of the matching string:

```
> s.search(/j.*a/i);
5
```

replace()

`replace()` allows you to replace the matched text with some other string. The following example removes all capital letters (it replaces them with blank strings):

```
> s.replace(/[A-Z]/g, '');
"elloavacriptorld"
```

If you omit the `g` modifier, you're only going to replace the first match:

```
> s.replace(/[A-Z]/, '');
"elloJavaScriptWorld"
```

When a match is found, if you want to include the matched text in the replacement string, you can access it using `$&`. Here's how to add an underscore before the match while keeping the match:

```
> s.replace(/[A-Z]/g, "_$&");
"_Hello_Java_Script_World"
```

When the regular expression contains groups (denoted by parentheses), the matches of each group are available as `$1` for the first group, `$2` the second, and so on.

```
> s.replace(/([A-Z])/g, "_$1");
"_Hello_Java_Script_World"
```

Imagine you have a registration form on your web page that asks for an e-mail address, username, and password. The user enters their e-mail, and then your JavaScript kicks in and suggests the username, taking it from the e-mail address:

```
> var email = "stoyan@phpied.com";
> var username = email.replace(/(.*)@.*/, "$1");
> username;
"stoyan"
```

Replace callbacks

When specifying the replacement, you can also pass a function that returns a string. This gives you the ability to implement any special logic you may need before specifying the replacements:

```
> function replaceCallback(match) {
    return "_" + match.toLowerCase();
```

```
  }

> s.replace(/[A-Z]/g, replaceCallback);
```
"_hello_java_script_world"

The callback function receives a number of parameters (the previous example ignores all but the first one):

- The first parameter is the match
- The last is the string being searched
- The one before last is the position of the match
- The rest of the parameters contain any strings matched by any groups in your regex pattern

Let's test this. First, let's create a variable to store the entire `arguments` array passed to the callback function:

```
> var glob;
```

Next, define a regular expression that has three groups and matches e-mail addresses in the format `something@something.something`:

```
> var re = /(.*)@(.*)\.(.*)/;
```

Finally, let's define a callback function that stores the arguments in `glob` and then returns the replacement:

```
var callback = function () {
  glob = arguments;
  return arguments[1] + ' at ' +
        arguments[2] + ' dot ' +
        arguments[3];
};
```

Now perform a test:

```
> "stoyan@phpied.com".replace(re, callback);
```
"stoyan at phpied dot com"

Here's what the callback function received as arguments:

```
> glob;
```
["stoyan@phpied.com", "stoyan", "phpied", "com", 0, "stoyan@phpied.com"]

split()

You already know about the `split()` method, which creates an array from an input string and a delimiter string. Let's take a string of comma-separated values and split it:

```
> var csv = 'one, two,three ,four';
> csv.split(',');
["one", " two", "three ", "four"]
```

Because the input string happens to have random inconsistent spaces before and after the commas, the array result has spaces too. With a regular expression, you can fix this using `\s*`, which means "zero or more spaces":

```
> csv.split(/\s*,\s*/);
["one", "two", "three", "four"]
```

Passing a string when a RegExp is expected

One last thing to note is that the four methods that you have just seen (`split()`, `match()`, `search()`, and `replace()`) can also take strings as opposed to regular expressions. In this case, the string argument is used to produce a new regex as if it was passed to `new RegExp()`.

An example of passing a string to `replace` is shown as follows:

```
> "test".replace('t', 'r');
"rest"
```

The above is the same as:

```
> "test".replace(new RegExp('t'), 'r');
"rest"
```

When you pass a string, you cannot set modifiers the way you do with a normal constructor or regex literal. There's a common source of errors when using a string instead of a regular expression object for string replacements, and it's due to the fact that the `g` modifier is `false` by default. The outcome is that only the first string is replaced, which is inconsistent with most other languages and a little confusing. For example:

```
> "pool".replace('o', '*');
"p*ol"
```

Most likely, you want to replace all occurrences:

```
> "pool".replace(/o/g, '*');
"p**l"
```

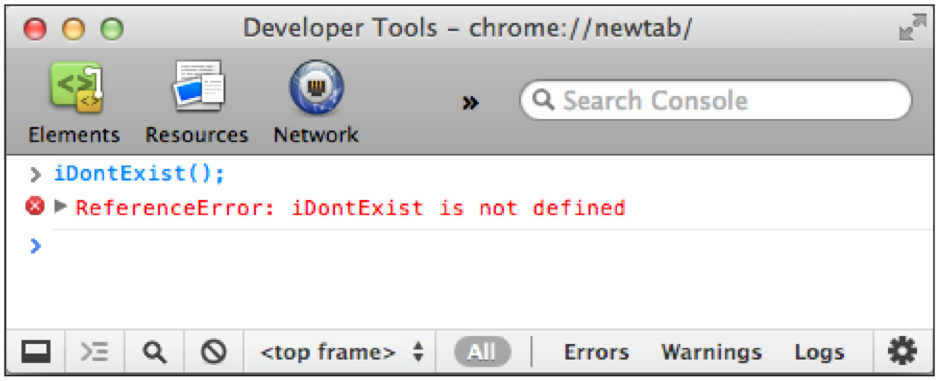
Error objects

Errors happen, and it's good to have the mechanisms in place so that your code can realize that there has been an error condition and can recover from it in a graceful manner. JavaScript provides the statements `try`, `catch`, and `finally` to help you deal with errors. If an error occurs, an error object is thrown. Error objects are created by using one of these built-in constructors: `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, and `URIError`. All of these constructors inherit from `Error`.

Let's just cause an error and see what happens. What's a simple way to cause an error? Just call a function that doesn't exist. Type this into the console:

```
> iDontExist();
```

You'll get something like this:



```
> iDontExist();
ReferenceError: iDontExist is not defined
```

The display of errors can vary greatly between browsers and other host environments. In fact, most recent browsers tend to hide the errors from the users. However, you cannot assume that all of your users have disabled the display of errors, and it is your responsibility to ensure an error-free experience for them. The previous error propagated to the user because the code didn't try to trap (catch) this error. The code didn't expect the error and was not prepared to handle it. Fortunately, it's trivial to trap the error. All you need is the `try` statement followed by a `catch` statement.

This code hides the error from the user:

```
try {
  iDontExist();
} catch (e) {
  // do nothing
}
```

Here you have:

- The `try` statement followed by a block of code
- The `catch` statement followed by a variable name in parentheses and another block of code

There can be an optional `finally` statement (not used in this example) followed by a block of code, which is executed regardless of whether there was an error or not.

In the previous example, the code block that follows the `catch` statement didn't do anything, but this is the place where you put the code that can help recover from the error, or at least give feedback to the user that your application is aware that there was a special condition.

The variable `e` in the parentheses after the `catch` statement contains an error object. Like any other object, it contains properties and methods. Unfortunately, different browsers implement these methods and properties differently, but there are two properties that are consistently implemented—`e.name` and `e.message`.

Let's try this code now:

```
try {
  iDontExist();
} catch (e) {
  alert(e.name + ': ' + e.message);
} finally {
  alert('Finally!');
}
```

This will present an `alert()` showing `e.name` and `e.message` and then another `alert()` saying **Finally!**.

In Firefox and Chrome, the first alert will say **ReferenceError: iDontExist is not defined**. In Internet Explorer, it will be **TypeError: Object expected**. This tells us two things:

- `e.name` contains the name of the constructor that was used to create the error object
- Because the error objects are not consistent across host environments (browsers), it would be somewhat tricky to have your code act differently depending on the type of error (the value of `e.name`)

You can also create error objects yourself using `new Error()` or any of the other error constructors, and then let the JavaScript engine know that there's an erroneous condition using the `throw` statement.

For example, imagine a scenario where you call the `maybeExists()` function and after that make calculations. You want to trap all errors in a consistent way, no matter whether the error is that `maybeExists()` doesn't exist or that your calculations found a problem. Consider this code:

```
try {
  var total = maybeExists();
  if (total === 0) {
    throw new Error('Division by zero!');
  } else {
    alert(50 / total);
  }
} catch (e) {
  alert(e.name + ': ' + e.message);
} finally {
  alert('Finally!');
}
```

This code will `alert()` different messages depending on whether or not `maybeExists()` is defined and the values it returns:

- If `maybeExists()` doesn't exist, you get **ReferenceError: maybeExists() is not defined** in Firefox and **TypeError: Object expected** in IE
- If `maybeExists()` returns 0, you'll get **Error: Division by zero!**
- If `maybeExists()` returns 2, you'll get an alert that says **25**

In all cases, there will be a second alert that says **Finally!**.

Instead of throwing a generic error, `throw new Error('Division by zero!')`, you can be more specific if you choose to, for example, throw `throw new RangeError('Division by zero!')`. Alternatively, you don't need a constructor, you can simply throw a normal object:

```
throw {
  name: "MyError",
  message: "OMG! Something terrible has happened"
}
```

This gives you cross-browser control over the error name.

Summary

In *Chapter 2, Primitive Data Types, Arrays, Loops, and Conditions*, you saw that there are five primitive data types (number, string, Boolean, null, and undefined) and we also said that everything that is not a primitive piece of data is an object. Now you also know that:

- Objects are like arrays, but you specify the keys.
- Objects contain properties.
- Properties can be functions (functions are data; remember `var f = function () {};`). Properties that are functions are also called methods.
- Arrays are actually objects with predefined numeric properties and an auto-incrementing `length` property.
- Array objects have a number of convenient methods (such as `sort()` or `slice()`).
- Functions are also objects and they have properties (such as `length` and `prototype`) and methods (such as `call()` and `apply()`).

Regarding the five primitive data types, apart from `undefined` and `null`, the other three have the corresponding constructor functions: `Number()`, `String()`, and `Boolean()`. Using these, you can create objects, called wrapper objects, which contain methods for working with primitive data elements.

`Number()`, `String()`, and `Boolean()` can be invoked:

- With the `new` operator—to create new objects
- Without the `new` operator—to convert any value to the corresponding primitive data type

Other built-in constructor functions you're now familiar with include: `Object()`, `Array()`, `Function()`, `Date()`, `RegExp()`, and `Error()`. You're also familiar with `Math`: a global object that is not a constructor.

Now you can see how objects have a central role in JavaScript programming, as pretty much everything is an object or can be wrapped by an object.

Finally, let's wrap up the literal notations you're now familiar with:

Name	**Literal**	**Constructor**	**Example**
Object	{}	`new Object()`	`{prop: 1}`
Array	[]	`new Array()`	`[1,2,3,'test']`
Regular expression	/pattern/ modifiers	`new RegExp('pattern', 'modifiers')`	`/java.*/img`

Exercises

1. Look at this code:

   ```
   function F() {
     function C() {
       return this;
     }
     return C();
   }
   var o = new F();
   ```

   Does the value of `this` refer to the global object or the object `o`?

2. What's the result of executing this piece of code?

   ```
   function C(){
     this.a = 1;
     return false;
   }
   console.log(typeof new C());
   ```

3. What's the result of executing the following piece of code?

   ```
   > c = [1, 2, [1, 2]];
   > c.sort();
   > c.join('--');
   > console.log(c);
   ```

4. Imagine the `String()` constructor didn't exist. Create a constructor function, `MyString()`, that acts like `String()` as closely as possible. You're not allowed to use any built-in string methods or properties, and remember that `String()` doesn't exist. You can use this code to test your constructor:

   ```
   > var s = new MyString('hello');
   > s.length;
     5

   > s[0];
     "h"

   > s.toString();
     "hello"

   > s.valueOf();
     "hello"

   > s.charAt(1);
     "e"
   ```

```
> s.charAt('2');
  "l"

> s.charAt('e');
  "h"

> s.concat(' world!');
  "hello world!"

> s.slice(1, 3);
  "el"

> s.slice(0, -1);
  "hell"

> s.split('e');
  ["h", "llo"]

> s.split('l');
  ["he", "", "o"]
```

> You can use a `for` loop to loop through the input string, treating it as an array.

5. Update your `MyString()` constructor to include a `reverse()` method.

> Try to leverage the fact that arrays have a `reverse()` method.

6. Imagine `Array()` doesn't exist and the array literal notation doesn't exist either. Create a constructor called `MyArray()` that behaves as close to `Array()` as possible. Test it with the following code:

```
> var a = new MyArray(1, 2, 3, "test");
> a.toString();
  "1,2,3,test"

> a.length;
  4

> a[a.length - 1];
  "test"

> a.push('boo');
  5

> a.toString();
  "1,2,3,test,boo"
```

```
> a.pop();
  "boo"

> a.toString();
  "1,2,3,test"

> a.join(',');
  "1,2,3,test"

> a.join(' isn\'t ');
  "1 isn't 2 isn't 3 isn't test"
```

If you found this exercise amusing, don't stop with the `join()` method; go on with as many methods as possible.

7. Imagine `Math` didn't exist. Create a `MyMath` object that also provides additional methods:
   - `MyMath.rand(min, max, inclusive)`—generates a random number between `min` and `max`, inclusive if `inclusive` is `true` (default)
   - `MyMath.min(array)`—returns the smallest number in a given array
   - `MyMath.max(array)`—returns the largest number in a given array

5 Prototype

In this chapter, you'll learn about the prototype property of the function objects. Understanding how the prototype works is an important part of learning the JavaScript language. After all, JavaScript is often classified as having a prototype-based object model. There's nothing particularly difficult about the prototype, but it's a new concept, and as such may sometimes take a bit of time to sink in. Like closures (see *Chapter 3*, *Functions*), the prototype is one of those things in JavaScript, which once you "get", they seem so obvious and make perfect sense. As with the rest of the book, you're strongly encouraged to type in and play around with the examples—this makes it much easier to learn and remember the concepts.

The following topics are discussed in this chapter:

- Every function has a `prototype` property and it contains an object
- Adding properties to the prototype object
- Using the properties added to the prototype
- The difference between own properties and properties of the prototype
- `__proto__`, the secret link every object keeps to its prototype
- Methods such as `isPrototypeOf()`, `hasOwnProperty()`, and `propertyIsEnumerable()`
- Enhancing built-in objects, such as arrays or strings (and why that can be a bad idea)

The prototype property

The functions in JavaScript are objects, and they contain methods and properties. Some of the methods that you're already familiar with are `apply()` and `call()`, and some of the other properties are `length` and `constructor`. Another property of the function objects is `prototype`.

If you define a simple function, `foo()`, you can access its properties as you would do with any other object.

```
> function foo(a, b) {
    return a * b;
  }
> foo.length;
2

> foo.constructor;
function Function() { [native code] }
```

The `prototype` property is a property that is available to you as soon as you define the function. Its initial value is an "empty" object.

```
> typeof foo.prototype;
"object"
```

It's as if you added this property yourself as follows:

```
> foo.prototype = {};
```

You can augment this empty object with properties and methods. They won't have any effect on the `foo()` function itself; they'll only be used if you call `foo()` as a constructor.

Adding methods and properties using the prototype

In the previous chapter, you learned how to define constructor functions that you can use to create (construct) new objects. The main idea is that inside a function invoked with `new`, you have access to the value `this`, which refers to the object to be returned by the constructor. Augmenting (adding methods and properties to) `this` is how you add functionality to the object being constructed.

Let's take a look at the constructor function `Gadget()`, which uses `this` to add two properties and one method to the objects it creates.

```
function Gadget(name, color) {
  this.name = name;
  this.color = color;
  this.whatAreYou = function () {
    return 'I am a ' + this.color + ' ' + this.name;
  };
}
```

Adding methods and properties to the `prototype` property of the constructor function is another way to add functionality to the objects this constructor produces. Let's add two more properties, `price` and `rating`, as well as a `getInfo()` method. Since `prototype` already points to an object, you can just keep adding properties and methods to it as follows:

```
Gadget.prototype.price = 100;
Gadget.prototype.rating = 3;
Gadget.prototype.getInfo = function () {
  return 'Rating: ' + this.rating +
         ', price: ' + this.price;
};
```

Alternatively, instead of adding properties to the `prototype` object one by one, you can overwrite the prototype completely, replacing it with an object of your choice.

```
Gadget.prototype = {
  price: 100,
  rating: ...  /* and so on... */
};
```

Using the prototype's methods and properties

All the methods and properties you have added to the prototype are available as soon as you create a new object using the constructor. If you create a `newtoy` object using the `Gadget()` constructor, you can access all the methods and properties already defined.

```
> var newtoy = new Gadget('webcam', 'black');
> newtoy.name;
"webcam"
```

```
> newtoy.color;
"black"

> newtoy.whatAreYou();
"I am a black webcam"

> newtoy.price;
100

> newtoy.rating;
3

> newtoy.getInfo();
"Rating: 3, price: 100"
```

It's important to note that the prototype is "live". Objects are passed by reference in JavaScript, and therefore, the prototype is not copied with every new object instance. What does this mean in practice? It means that you can modify the prototype at any time and all the objects (even those created before the modification) will "see" the changes.

Let's continue the example by adding a new method to the prototype:

```
Gadget.prototype.get = function (what) {
  return this[what];
};
```

Even though `newtoy` was created *before* the `get()` method was defined, `newtoy` still has access to the new method:

```
> newtoy.get('price');
100

> newtoy.get('color');
"black"
```

Own properties versus prototype properties

In the preceding example, `getInfo()` was used internally to access the properties of the object. It could've also used `Gadget.prototype` to achieve the same output.

```
Gadget.prototype.getInfo = function () {
  return 'Rating: ' + Gadget.prototype.rating +
         ', price: ' + Gadget.prototype.price;
};
```

What's the difference? To answer this question, let's examine how the prototype works in more detail.

Let's take the `newtoy` object again.

```
var newtoy = new Gadget('webcam', 'black');
```

When you try to access a property of `newtoy`, say, `newtoy.name`, the JavaScript engine looks through all of the properties of the object searching for one called `name`, and if it finds it, it returns its value.

```
> newtoy.name;
"webcam"
```

What if you try to access the `rating` property? The JavaScript engine examines all of the properties of `newtoy` and doesn't find the one called `rating`. Then, the script engine identifies the prototype of the constructor function used to create this object (the same as if you do `newtoy.constructor.prototype`). If the property is found in the prototype object, the following property is used:

```
> newtoy.rating;
3
```

You can do the same and access the prototype directly. Every object has a constructor property, which is a reference to the function that created the object, so in this case:

```
> newtoy.constructor === Gadget;
true

> newtoy.constructor.prototype.rating;
3
```

Now, let's take this lookup one step further. Every object has a constructor. The prototype is an object, so it must have a constructor too, which, in turn, has a prototype. You can go up the *prototype chain* and you will eventually end up with the built-in `Object()` object, which is the highest-level parent. In practice, this means that if you try `newtoy.toString()` and `newtoy` doesn't have its own `toString()` method and its prototype doesn't either, in the end you'll get the object's `toString()` method:

```
> newtoy.toString();
"[object Object]"
```

Overwriting a prototype's property with an own property

As the above discussion demonstrates, if one of your objects doesn't have a certain property of its own, it can use one (if it exists) somewhere up the prototype chain. What if the object does have its own property and the prototype also has one with the same name? Then, the own property takes precedence over the prototype's.

Consider a scenario where a property name exists as both an own property and a property of the prototype object.

```
> function Gadget(name) {
    this.name = name;
  }
> Gadget.prototype.name = 'mirror';
```

Creating a new object and accessing its `name` property gives you the object's own `name` property.

```
> var toy = new Gadget('camera');
> toy.name;
"camera"
```

You can tell where the property was defined by using `hasOwnProperty()`.

```
> toy.hasOwnProperty('name');
true
```

If you delete the toy object's own `name` property, the prototype's property with the same name "shines through".

```
> delete toy.name;
true

> toy.name;
"mirror"

> toy.hasOwnProperty('name');
false
```

Of course, you can always recreate the object's own property.

```
> toy.name = 'camera';
> toy.name;
"camera"
```

You can play around with the method `hasOwnProperty()` to find out the origins of a particular property you're curious about. The method `toString()` was mentioned earlier. Where is it coming from?

```
> toy.toString();
"[object Object]"

> toy.hasOwnProperty('toString');
false
```

```
> toy.constructor.hasOwnProperty('toString');
false

> toy.constructor.prototype.hasOwnProperty('toString');
false

> Object.hasOwnProperty('toString');
false

> Object.prototype.hasOwnProperty('toString');
true
```

Ahaa!

Enumerating properties

If you want to list all the properties of an object, you can use a `for-in` loop. In *Chapter 2*, *Primitive Data Types, Arrays, Loops, and Conditions*, you saw that you can also loop through all the elements of an array with `for-in`, but as mentioned there, `for` is better suited for arrays and `for-in` is for objects. Let's take an example of constructing a query string for a URL from an object:

```
var params = {
  productid: 666,
  section: 'products'
};

var url = 'http://example.org/page.php?',
    i,
    query = [];

for (i in params) {
  query.push(i + '=' + params[i]);
}

url += query.join('&');
```

This produces the `url` string as follows:

```
"http://example.org/page.php?productid=666&section=products"
```

There are a few details to be aware of:

- Not all properties show up in a `for-in` loop. For example, the length (for arrays) and constructor properties don't show up. The properties that do show up are called **enumerable**. You can check which ones are enumerable with the help of the `propertyIsEnumerable()` method that every object provides. In ES5, you can specify which properties are enumerable, while in ES3 you don't have that control.
- Prototypes that come through the prototype chain also show up, provided they are enumerable. You can check if a property is an object's own property or a prototype's property using the `hasOwnProperty()` method.
- `propertyIsEnumerable()` returns `false` for all of the prototype's properties, even those that are enumerable and show up in the `for-in` loop.

Let's see these methods in action. Take this simplified version of `Gadget()`:

```
function Gadget(name, color) {
  this.name = name;
  this.color = color;
  this.getName = function () {
    return this.name;
  };
}
Gadget.prototype.price = 100;
Gadget.prototype.rating = 3;
```

Create a new object as follows:

```
var newtoy = new Gadget('webcam', 'black');
```

Now, if you loop using a `for-in` loop, you see all of the object's properties, including those that come from the prototype:

```
for (var prop in newtoy) {
  console.log(prop + ' = ' + newtoy[prop]);
}
```

The result also contains the object's methods (since methods are just properties that happen to be functions):

```
name = webcam
color = black
getName = function () {
 return this.name;
}
price = 100
rating = 3
```

If you want to distinguish between the object's own properties and the prototype's properties, use `hasOwnProperty()`. Try the following first:

```
> newtoy.hasOwnProperty('name');
true

> newtoy.hasOwnProperty('price');
false
```

Let's loop again, but this time showing only the object's own properties.

```
for (var prop in newtoy) {
  if (newtoy.hasOwnProperty(prop)) {
    console.log(prop + '=' + newtoy[prop]);
  }
}
```

The result is as follows:

```
name=webcam
color=black
getName = function () {
 return this.name;
}
```

Now let's try `propertyIsEnumerable()`. This method returns `true` for the object's own properties that are not built-in.

```
> newtoy.propertyIsEnumerable('name');
true
```

Most built-in properties and methods are not enumerable.

```
> newtoy.propertyIsEnumerable('constructor');
false
```

Any properties coming down the prototype chain are not enumerable.

```
> newtoy.propertyIsEnumerable('price');
false
```

Note, however, that such properties are enumerable if you reach the object contained in the prototype and invoke its `propertyIsEnumerable()` method.

```
> newtoy.constructor.prototype.propertyIsEnumerable('price');
true
```

isPrototypeOf()

Objects also have the `isPrototypeOf()` method. This method tells you whether that specific object is used as a prototype of another object.

Let's take a simple object named `monkey`.

```
var monkey = {
  hair: true,
  feeds: 'bananas',
  breathes: 'air'
};
```

Now let's create a `Human()` constructor function and set its prototype property to point to `monkey`.

```
function Human(name) {
  this.name = name;
}
Human.prototype = monkey;
```

Now if you create a new `Human` object called `george` and ask "is `monkey` the prototype of `george`?", you'll get `true`.

```
> var george = new Human('George');
> monkey.isPrototypeOf(george);
true
```

Note that you have to know, or suspect, who the prototype is and then ask "is it true that your `prototype` is `monkey`?" in order to confirm your suspicion. But what if you don't suspect anything and you have no idea? Can you just ask the object to tell you its prototype? The answer is you can't in all browsers, but you can in most of them. Most recent browsers have implemented the addition to ES5 called `Object.getPrototypeOf()`.

```
> Object.getPrototypeOf(george).feeds;
"bananas"

> Object.getPrototypeOf(george) === monkey;
true
```

For some of the pre-ES5 environments that don't have `getPrototypeOf()`, you can use the special property `__proto__`.

The secret __proto__ link

As you already know, the `prototype` property is consulted when you try to access a property that does not exist in the current object.

Consider another object called `monkey` and use it as a prototype when creating objects with the `Human()` constructor.

```
> var monkey = {
    feeds: 'bananas',
    breathes: 'air'
  };
> function Human() {}
> Human.prototype = monkey;
```

Now, let's create a `developer` object and give it some properties.

```
> var developer = new Human();
> developer.feeds = 'pizza';
> developer.hacks = 'JavaScript';
```

Now let's access these properties. For example, `hacks` is a property of the `developer` object.

```
> developer.hacks;
"JavaScript"
```

`feeds` could also be found in the object.

```
> developer.feeds;
"pizza"
```

`breathes` doesn't exist as a property of the `developer` object, so the prototype is looked up, as if there is a secret link, or a secret passageway, that leads to the prototype object.

```
> developer.breathes;
"air"
```

The secret link is exposed in most modern JavaScript environments as the `__proto__` property (the word "proto" with two underscores before and two after).

```
> developer.__proto__ === monkey;
true
```

You can use this secret property for learning purposes, but it's not a good idea to use it in your real scripts because it does not exist in all browsers (notably Internet Explorer), so your scripts won't be portable.

Be aware that `__proto__` is not the same as `prototype`, since `__proto__` is a property of the instances (objects), whereas `prototype` is a property of the constructor functions used to create those objects.

```
> typeof developer.__proto__;
"object"

> typeof developer.prototype;
"undefined"

> typeof developer.constructor.prototype;
"object"
```

Once again, you should use `__proto__` only for learning or debugging purposes. Or, if you're lucky enough and your code only needs to work in ES5-compliant environments, you can use `Object.getPrototypeOf()`.

Augmenting built-in objects

The objects created by the built-in constructor functions such as `Array`, `String`, and even `Object` and `Function` can be augmented (or enhanced) through the use of prototypes. This means that you can, for example, add new methods to the `Array` prototype, and in this way you can make them available to all arrays. Let's see how to do this.

In PHP, there is a function called `in_array()`, which tells you if a value exists in an array. In JavaScript, there is no `inArray()` method (although in ES5 there's `indexOf()`, which you can use for the same purpose). So, let's implement it and add it to `Array.prototype`.

```
Array.prototype.inArray = function (needle) {
  for (var i = 0, len = this.length; i < len; i++) {
    if (this[i] === needle) {
      return true;
    }
  }
  return false;
};
```

Now all arrays have access to the new method. Let's test this.

```
> var colors = ['red', 'green', 'blue'];
> colors.inArray('red');
true

> colors.inArray('yellow');
false
```

That was nice and easy! Let's do it again. Imagine your application often needs to spell words backwards and you feel there should be a built-in `reverse()` method for string objects. After all, arrays have `reverse()`. You can easily add a `reverse()` method to the `String` prototype by borrowing `Array.prototype.reverse()` (there was a similar exercise at the end of *Chapter 4*, *Objects*).

```
String.prototype.reverse = function () {
  return Array.prototype.reverse.
             apply(this.split('')).join('');
};
```

This code uses `split()` to create an array from a string, then calls the `reverse()` method on this array, which produces a reversed array. The resulting array is then turned back into a string using `join()`. Let's test the new method.

```
> "bumblebee".reverse();
"eebelbmub"
```

That is a nice name for a big and scary (and potentially hairy) mythical creature, isn't it?

Augmenting built-in objects – discussion

Augmenting built-in objects through the prototype is a powerful technique, and you can use it to shape JavaScript in any way you like. Because of its power, though, you should always thoroughly consider your options before using this approach.

The reason is that once you know JavaScript, you're expecting it to work the same way, no matter which third-party library or widget you're using. Modifying core objects could confuse the users and maintainers of your code and create unexpected errors.

JavaScript evolves and browser's vendors continuously support more features. What you consider a missing method today and decide to add to a core prototype could be a built-in method tomorrow. In this case, your method is no longer needed. Additionally, what if you have already written a lot of code that uses the method and your method is slightly different from the new built-in implementation?

The most common and acceptable use case for augmenting built-in prototypes is to add support for new features (ones that are already standardized by the ECMAScript committee and implemented in new browsers) to old browsers. One example would be adding an ES5 method to old versions of IE. These extensions are known as **shims** or **polyfills**.

When augmenting prototypes, you first check if the method exists before implementing it yourself. This way, you use the native implementation in the browser if one exists. For example, let's add the `trim()` method for strings, which is a method that exists in ES5 but is missing in older browsers.

```
if (typeof String.prototype.trim !== 'function') {
  String.prototype.trim = function () {
    return this.replace(/^\s+|\s+$/g,'');
  };
}

> " hello ".trim();
"hello"
```

Best practice
If you decide to augment a built-in object or its prototype with a new property, do check for the existence of the new property first.

Prototype gotchas

There are two important behaviors to consider when dealing with prototypes:

- The prototype chain is live except when you completely replace the prototype object
- `prototype.constructor` is not reliable

Let's create a simple constructor function and two objects.

```
> function Dog() {
    this.tail = true;
  }
> var benji = new Dog();
> var rusty = new Dog();
```

Even after you've created the objects `benji` and `rusty`, you can still add properties to the prototype of `Dog()` and the existing objects will have access to the new properties. Let's throw in the method `say()`.

```
> Dog.prototype.say = function () {
    return 'Woof!';
  };
```

Both objects have access to the new method.

```
> benji.say();
"Woof!"

 rusty.say();
"Woof!"
```

Up to this point, if you consult your objects asking which constructor function was used to create them, they'll report it correctly.

```
> benji.constructor === Dog;
true

> rusty.constructor === Dog;
true
```

Now, let's completely overwrite the prototype object with a brand new object.

```
> Dog.prototype = {
    paws: 4,
    hair: true
  };
```

It turns out that the old objects do not get access to the new prototype's properties; they still keep the secret link pointing to the old prototype object.

```
> typeof benji.paws;
"undefined"

> benji.say();
"Woof!"

> typeof benji.__proto__.say;
"function"

> typeof benji.__proto__.paws;
"undefined"
```

Any new objects you create from now on will use the updated prototype.

```
> var lucy = new Dog();
> lucy.say();
TypeError: lucy.say is not a function

> lucy.paws;
4
```

The secret `__proto__` link points to the new prototype object.

```
> typeof lucy.__proto__.say;
"undefined"

> typeof lucy.__proto__.paws;
"number"
```

Now the `constructor` property of the new object no longer reports correctly. You would expect it to point to `Dog()`, but instead it points to `Object()`.

```
> lucy.constructor;
function Object() { [native code] }

> benji.constructor;
function Dog() {
  this.tail = true;
}
```

You can easily prevent this confusion by resetting the `constructor` property after you overwrite the prototype completely.

```
> function Dog() {}
> Dog.prototype = {};
> new Dog().constructor === Dog;
false

> Dog.prototype.constructor = Dog;
> new Dog().constructor === Dog;
true
```

Best practice

When you overwrite the prototype, remember to reset the `constructor` property.

Summary

Let's summarize the most important topics you have learned in this chapter:

- All functions have a property called `prototype`. Initially it contains an "empty" object (an object without any own properties).
- You can add properties and methods to the prototype object. You can even replace it completely with an object of your choice.
- When you create an object using a function as a constructor (with `new`), the object gets a secret link pointing to the prototype of the constructor, and can access the prototype's properties.
- An object's own properties take precedence over a prototype's properties with the same name.
- Use the method `hasOwnProperty()` to differentiate between an object's own properties and prototype properties.
- There is a prototype chain. When you execute `foo.bar`, and if your object `foo` doesn't have a property called `bar`, the JavaScript interpreter looks for a `bar` property in the prototype. If none is found, it keeps searching in the prototype's prototype, then the prototype of the prototype's prototype, and it will keep going all the way up to `Object.prototype`.
- You can augment the prototypes of built-in constructor functions and all objects will see your additions. Assign a function to `Array.prototype.flip` and all arrays will immediately get a `flip()` method, as in `[1,2,3].flip()`. But do check if the method/property you want to add already exists, so you can future-proof your scripts.

Exercises

1. Create an object called `shape` that has the type `property` and a `getType()` method.
2. Define a `Triangle()` constructor function whose prototype is `shape`. Objects created with `Triangle()` should have three own properties—`a`, `b`, and `c`, representing the lengths of the sides of a triangle.
3. Add a new method to the prototype called `getPerimeter()`.

4. Test your implementation with the following code:

   ```
   > var t = new Triangle(1, 2, 3);
   > t.constructor === Triangle;
          true

   > shape.isPrototypeOf(t);
           true

   > t.getPerimeter();
         6

   > t.getType();
          "triangle"
   ```

5. Loop over `t` showing only own properties and methods (none of the prototype's).
6. Make the following code work:

   ```
   > [1, 2, 3, 4, 5, 6, 7, 8, 9].shuffle();
          [2, 4, 1, 8, 9, 6, 5, 3, 7]
   ```

6
Inheritance

In this chapter, let's focus on the inheritance part. This is one of the most interesting features, as it allows you to reuse existing code, thus promoting laziness, which is likely to be what brought human species to computer programming in the first place.

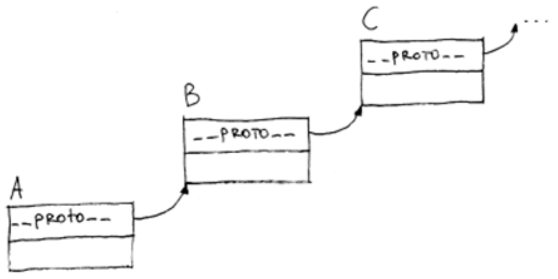
JavaScript is a dynamic language and there is usually more than one way to achieve any given task. Inheritance is not an exception. In this chapter, you'll see some common patterns for implementing inheritance. Having a good understanding of these patterns will help you pick the right one, or the right mix, depending on your task, project or your style.

Prototype chaining

Let's start with the default way of implementing inheritance—inheritance chaining through the prototype.

As you already know, every function has a `prototype` property, which points to an object. When a function is invoked using the `new` operator, an object is created and returned. This new object has a secret link to the prototype object. The secret link (called `__proto__` in some environments) allows methods and properties of the prototype object to be used as if they belonged to the newly-created object.

The prototype object is just a regular object and, therefore, it also has the secret link to its prototype. And so a chain is created, called a **prototype chain**:



In this illustration, an object A contains a number of properties. One of the properties is the hidden `__proto__` property, which points to another object, B. B's `__proto__` property points to C. This chain ends with the `Object.prototype` object—the grandparent, and every object inherits from it.

This is all good to know, but how does it help you? The practical side is that when object A lacks a property but B has it, A can still access this property as its own. The same applies if B also doesn't have the required property, but C does. This is how inheritance takes place: an object can access any property found somewhere down the inheritance chain.

Throughout the rest of this chapter, you'll see different examples that use the following hierarchy: a generic Shape parent is inherited by a 2D shape, which in turn is inherited by any number of specific two-dimensional shapes such as a Triangle, Rectangle, and so on.

Prototype chaining example

Prototype chaining is the default way to implement inheritance. In order to implement the hierarchy, let's define three constructor functions.

```
function Shape(){
this.name = 'Shape';
this.toString = function () {
return this.name;
};
}

function TwoDShape(){
this.name = '2D shape';
}

function Triangle(side, height){
this.name = 'Triangle';
this.side = side;
this.height = height;
this.getArea = function () {
return this.side * this.height / 2;
};
}
```

The code that performs the inheritance magic is as follows:

```
TwoDShape.prototype = new Shape();
Triangle.prototype = new TwoDShape();
```

What's happening here? You take the object contained in the `prototype` property of `TwoDShape` and instead of augmenting it with individual properties, you completely overwrite it with another object, created by invoking the `Shape()` constructor with `new`. The same for `Triangle`: its prototype is replaced with an object created by `new TwoDShape()`. It's important to remember that JavaScript works with objects, not classes. You need to create an instance using the `new Shape()` constructor and after that you can inherit its properties; you don't inherit from `Shape()` directly. Additionally, after inheriting, you can modify the `Shape()` constructor, overwrite it, or even delete it, and this will have no effect on `TwoDShape`, because all you needed is one instance to inherit from.

As you know from the previous chapter, overwriting the prototype (as opposed to just adding properties to it), has side effects on the `constructor` property. Therefore, it's a good idea to reset the `constructor` after inheriting:

```
TwoDShape.prototype.constructor = TwoDShape;
Triangle.prototype.constructor = Triangle;
```

Now, let's test what has happened so far. Creating a `Triangle` object and calling its own `getArea()` method works as expected:

```
>var my = new Triangle(5, 10);
>my.getArea();
25
```

Although the `my` object doesn't have its own `toString()` method, it inherited one and you can call it. Note, how the inherited method `toString()` binds the `this` object to `my`.

```
>my.toString();
"Triangle"
```

It's fascinating to consider what the JavaScript engine does when you call `my.toString()`:

- It loops through all of the properties of `my` and doesn't find a method called `toString()`.
- It looks at the object that `my.__proto__` points to; this object is the instance `new TwoDShape()` created during the inheritance process.
- Now, the JavaScript engine loops through the instance of `TwoDShape` and doesn't find a `toString()` method. It then checks the `__proto__` of that object. This time `__proto__` points to the instance created by `new Shape()`.
- The instance of `new Shape()` is examined and `toString()` is finally found.
- This method is invoked in the context of `my`, meaning that `this` points to `my`.

If you ask `my`, "who's your constructor?" it reports it correctly because of the reset of the `constructor` property after the inheritance:

```
>my.constructor === Triangle;
true
```

Using the `instanceof` operator you can validate that `my` is an instance of all three constructors.

```
> my instanceof Shape;
true

> my instanceofTwoDShape;
true

> my instanceof Triangle;
true

> my instanceof Array;
false
```

The same happens when you call `isPrototypeOf()` on the constructors passing `my`:

```
>Shape.prototype.isPrototypeOf(my);
true

>TwoDShape.prototype.isPrototypeOf(my);
true

>Triangle.prototype.isPrototypeOf(my);
true

>String.prototype.isPrototypeOf(my);
false
```

You can also create objects using the other two constructors. Objects created with `new TwoDShape()` also get the method `toString()`, inherited from `Shape()`.

```
>var td = new TwoDShape();
>td.constructor === TwoDShape;
true

>td.toString();
"2D shape"

>var s = new Shape();
>s.constructor === Shape;
true
```

Moving shared properties to the prototype

When you create objects using a constructor function, own properties are added using `this`. This could be inefficient in cases where properties don't change across instances. In the previous example, `Shape()` was defined like so:

```
function Shape(){
this.name = 'Shape';
}
```

This means that every time you create a new object using `new Shape()` a new `name` property is created and stored somewhere in the memory. The other option is to have the `name` property added to the prototype and shared among all the instances:

```
function Shape() {}
Shape.prototype.name = 'Shape';
```

Now, every time you create an object using `new Shape()`, this object doesn't get its own property `name`, but uses the one added to the prototype. This is more efficient, but you should only use it for properties that don't change from one instance to another. Methods are ideal for this type of sharing.

Let's improve on the preceding example by adding all methods and suitable properties to the `prototype`. In the case of `Shape()` and `TwoDShape()` everything is meant to be shared:

```
// constructor
function Shape() {}

// augment prototype
Shape.prototype.name = 'Shape';
Shape.prototype.toString = function () {
return this.name;
};

// another constructor
function TwoDShape() {}

// take care of inheritance
TwoDShape.prototype = new Shape();
TwoDShape.prototype.constructor = TwoDShape;

// augment prototype
TwoDShape.prototype.name = '2D shape';
```

As you can see, you have to take care of inheritance first before augmenting the prototype. Otherwise anything you add to `TwoDShape.prototype` gets wiped out when you inherit.

The `Triangle` constructor is a little different, because every object it creates is a new triangle, which is likely to have different dimensions. So it's good to keep `side` and `height` as own properties and share the rest. The method `getArea()`, for example, is the same regardless of the actual dimensions of each triangle. Again, you do the inheritance bit first and then augment the prototype.

```
function Triangle(side, height) {
this.side = side;
this.height = height;
}
// take care of inheritance
Triangle.prototype = new TwoDShape();
Triangle.prototype.constructor = Triangle;

// augment prototype
Triangle.prototype.name = 'Triangle';
Triangle.prototype.getArea = function () {
return this.side * this.height / 2;
};
```

All the preceding test code work exactly the same, for example:

```
>var my = new Triangle(5, 10);
>my.getArea();
25

>my.toString();
"Triangle"
```

There is only a slight behind-the-scenes difference when calling `my.toString()`. The difference is that there is one more lookup to be done before the method is found in the `Shape.prototype`, as opposed to in the `new Shape()` instance like it was in the previous example.

You can also play with `hasOwnProperty()` to see the difference between the own property versus a property coming down the prototype chain.

```
>my.hasOwnProperty('side');
true

>my.hasOwnProperty('name');
false
```

The calls to `isPrototypeOf()` and the `instanceof` operator from the previous example work exactly the same:

```
>TwoDShape.prototype.isPrototypeOf(my);
true

> my instanceof Shape;
true
```

Inheriting the prototype only

As explained previously, for reasons of efficiency you should add the reusable properties and methods to the prototype. If you do so, then it's a good idea to inherit only the prototype, because all the reusable code is there. This means that inheriting the `Shape.prototype` object is better than inheriting the object created with `new Shape()`. After all, `new Shape()` only gives you own shape properties that are not meant to be reused (otherwise they would be in the prototype). You gain a little more efficiency by:

- Not creating a new object for the sake of inheritance alone
- Having less lookups during runtime (when it comes to searching for `toString()` for example)

Here's the updated code; the changes are highlighted:

```
function Shape() {}
// augment prototype
Shape.prototype.name = 'Shape';
Shape.prototype.toString = function () {
```

```
return this.name;
};

function TwoDShape() {}
// take care of inheritance
TwoDShape.prototype = Shape.prototype;
TwoDShape.prototype.constructor = TwoDShape;
// augment prototype
TwoDShape.prototype.name = '2D shape';

function Triangle(side, height) {
this.side = side;
this.height = height;
}

// take care of inheritance
Triangle.prototype = TwoDShape.prototype;
Triangle.prototype.constructor = Triangle;
// augment prototype
Triangle.prototype.name = 'Triangle';
Triangle.prototype.getArea = function () {
return this.side * this.height / 2;
};
```

The test code gives you the same result:

```
>var my = new Triangle(5, 10);
>my.getArea();
25

>my.toString();
"Triangle"
```

What's the difference in the lookups when calling `my.toString()`? First, as usual, the JavaScript engine looks for a method `toString()` of the `my` object itself. The engine doesn't find such a method, so it inspects the prototype. The prototype turns out to be pointing to the same object that the prototype of `TwoDShape` points to and also the same object that `Shape.prototype` points to. Remember, that objects are not copied by value, but only by reference. So the lookup is only a two-step process as opposed to four (in the previous example) or three (in the first example).

Simply copying the prototype is more efficient but it has a side effect: because all the prototypes of the children and parents point to the same object, when a child modifies the prototype, the parents get the changes, and so do the siblings.

Look at this line:

```
Triangle.prototype.name = 'Triangle';
```

It changes the `name` property, so it effectively changes `Shape.prototype.name` too. If you create an instance using `new Shape()`, its `name` property says **"Triangle"**:

```
>var s = new Shape();
>s.name;
"Triangle"
```

This method is more efficient but may not suit all your use cases.

A temporary constructor – new F()

A solution to the previously outlined problem, where all prototypes point to the same object and the parents get children's properties, is to use an intermediary to break the chain. The intermediary is in the form of a temporary constructor function. Creating an empty function `F()` and setting its `prototype` to the prototype of the parent constructor, allows you to call `new F()` and create objects that have no properties of their own, but inherit everything from the parent's `prototype`.

Let's take a look at the modified code:

```
function Shape() {}
// augment prototype
Shape.prototype.name = 'Shape';
Shape.prototype.toString = function () {
return this.name;
};

function TwoDShape() {}
// take care of inheritance
var F = function () {};
F.prototype = Shape.prototype;
TwoDShape.prototype = new F();
TwoDShape.prototype.constructor = TwoDShape;
// augment prototype
TwoDShape.prototype.name = '2D shape';

function Triangle(side, height) {
this.side = side;
this.height = height;
}

// take care of inheritance
```

```
var F = function () {};
F.prototype = TwoDShape.prototype;
Triangle.prototype = new F();
Triangle.prototype.constructor = Triangle;
// augment prototype
Triangle.prototype.name = 'Triangle';
Triangle.prototype.getArea = function () {
return this.side * this.height / 2;
};
```

Creating `my` triangle and testing the methods:

```
>var my = new Triangle(5, 10);
>my.getArea();
25

>my.toString();
"Triangle"
```

Using this approach, the prototype chain stays in place:

```
>my.__proto__ === Triangle.prototype;
true

>my.__proto__.constructor === Triangle;
true

>my.__proto__.__proto__ === TwoDShape.prototype;
true

>my.__proto__.__proto__.__proto__.constructor === Shape;
true
```

And also the parents' properties are not overwritten by the children:

```
>var s = new Shape();
>s.name;
"Shape"

>"I am a " + new TwoDShape(); // calling toString()
"I am a 2D shape"
```

At the same time, this approach supports the idea that only properties and methods added to the prototype should be inherited, and own properties should not. The rationale behind this is that own properties are likely to be too specific to be reusable.

Uber – access to the parent from a child object

Classical OO languages usually have a special syntax that gives you access to the parent class, also referred to as superclass. This could be convenient when a child wants to have a method that does everything the parent's method does plus something in addition. In such cases, the child calls the parent's method with the same name and works with the result.

In JavaScript, there is no such special syntax, but it's trivial to achieve the same functionality. Let's rewrite the last example and, while taking care of inheritance, also create an `uber` property that points to the parent's prototype object.

```
function Shape() {}
// augment prototype
Shape.prototype.toString = function()
{
    var constr = this.constructor;
    return constr.uber
    ? constr.uber.toString() + ', ' + this.name
    :this.name;
};

function TwoDShape() {}
// take care of inheritance
var F = function () {};
F.prototype = Shape.prototype;
TwoDShape.prototype = new F();
TwoDShape.prototype.constructor = TwoDShape;
TwoDShape.uber = Shape.prototype;
// augment prototype
TwoDShape.prototype.name = '2D shape';

function Triangle(side, height) {
this.side = side;
this.height = height;
}

// take care of inheritance
var F = function () {};
F.prototype = TwoDShape.prototype;
Triangle.prototype = new F();
Triangle.prototype.constructor = Triangle;
Triangle.uber = TwoDShape.prototype;
```

```
// augment prototype
Triangle.prototype.name = 'Triangle';
Triangle.prototype.getArea = function () {
return this.side * this.height / 2;
};
```

The new things here are:

- A new`uber` property points to the parent's prototype
- The updated `toString()`method

Previously, `toString()` only returned `this.name`. Now, in addition to that, there is a check to see whether `this.constructor.uber` exists and, if it does, call its `toString()` first. `this.constructor` is the function itself, and `this.constructor.uber` points to the parent's `prototype`. The result is that when you call `toString()` for a `Triangle` instance, all `toString()` methods up the prototype chain are called:

```
>var my = new Triangle(5, 10);
>my.toString();
"Shape, 2D shape, Triangle"
```

The name of the property `uber` could've been "superclass" but this would suggest that JavaScript has classes. Ideally it could've been "super" (as in Java), but "super" is a reserved word in JavaScript. The German word "über" suggested by Douglass Crockford, means more or less the same as "super" and, you have to admit, it sounds uber-cool.

Isolating the inheritance part into a function

Let's move the code that takes care of all of the inheritance details from the last example into a reusable `extend()` function:

```
function extend(Child, Parent) {
var F = function () {};
F.prototype = Parent.prototype;
Child.prototype = new F();
Child.prototype.constructor = Child;
Child.uber = Parent.prototype;
}
```

Using this function (or your own custom version of it) helps you keep your code clean with regard to the repetitive inheritance-related tasks. This way you can inherit by simply using:

```
extend(TwoDShape, Shape);
```

and

```
extend(Triangle, TwoDShape);
```

Let's see a complete example:

```
// inheritance helper
function extend(Child, Parent) {
var F = function () {};
F.prototype = Parent.prototype;
Child.prototype = new F();
Child.prototype.constructor = Child;
Child.uber = Parent.prototype;
}

// define -> augment
function Shape() {}
Shape.prototype.name = 'Shape';
Shape.prototype.toString = function () {
  return this.constructor.uber
    ? this.constructor.uber.toString() + ', ' + this.name
    : this.name;
};

// define -> inherit -> augment
function TwoDShape() {}
extend(TwoDShape, Shape);
TwoDShape.prototype.name = '2D shape';


// define
function Triangle(side, height) {
this.side = side;
this.height = height;
}
// inherit
extend(Triangle, TwoDShape);
// augment
Triangle.prototype.name = 'Triangle';
Triangle.prototype.getArea = function () {
  return this.side * this.height / 2;
};
```

Testing:

```
> new Triangle().toString();
"Shape, 2D shape, Triangle"
```

Copying properties

Now, let's try a slightly different approach. Since inheritance is all about reusing code, can you simply copy the properties you like from one object to another? Or from a parent to a child? Keeping the same interface as the preceding `extend()` function, you can create a function `extend2()` which takes two constructor functions and copies all of the properties from the parent's prototype to the child's prototype. This will, of course, carry over methods too, as methods are just properties that happen to be functions.

```
function extend2(Child, Parent) {
var p = Parent.prototype;
var c = Child.prototype;
for (vari in p) {
c[i] = p[i];
}
c.uber = p;
}
```

As you can see, a simple loop through the properties is all it takes. As with the previous example, you can set an `uber` property if you want to have handy access to parent's methods from the child. Unlike the previous example though, it's not necessary to reset the `Child.prototype.constructor` because here the child prototype is augmented, not overwritten completely, so the `constructor` property points to the initial value.

This method is a little inefficient compared to the previous method because properties of the child prototype are being duplicated instead of simply being looked up via the prototype chain during execution. Bear in mind that this is only true for properties containing primitive types. All objects (including functions and arrays) are not duplicated, because these are passed by reference only.

Let's see an example of using two constructor functions, `Shape()` and `TwoDShape()`. The `Shape()` function's prototype object contains a primitive property, `name`, and a non-primitive one—the `toString()`method:

```
var Shape = function () {};
var TwoDShape = function () {};
Shape.prototype.name = 'Shape';
Shape.prototype.toString = function () {
  return this.uber
    ? this.uber.toString() + ', ' + this.name
    : this.name;
};
```

If you inherit with `extend()`, neither the objects created with `TwoDShape()` nor its prototype get an own `name` property, but they have access to the one they inherit.

```
> extend(TwoDShape, Shape);
>var td = new TwoDShape();
>td.name;
"Shape"

>TwoDShape.prototype.name;
"Shape"

>td.__proto__.name;
"Shape"

>td.hasOwnProperty('name');
false

> td.__proto__.hasOwnProperty('name');
false
```

But if you inherit with `extend2()`, the prototype of `TwoDShape()` gets its own copy of the `name` property. It also gets its own copy of `toString()`, but it's a reference only, so the function will not be recreated a second time.

```
>extend2(TwoDShape, Shape);
>var td = new TwoDShape();
> td.__proto__.hasOwnProperty('name');
true

> td.__proto__.hasOwnProperty('toString');
true

> td.__proto__.toString === Shape.prototype.toString;
true
```

As you can see, the two `toString()` methods are the same function object. This is good because it means that no unnecessary duplicates of the methods are created.

So, you can say that `extend2()` is less efficient than `extend()` because it recreates the properties of the prototype. But, this is not so bad because only the primitive data types are duplicated. Additionally, this is beneficial during the prototype chain lookups as there are fewer chain links to follow before finding the property.

Take a look at the `uber` property again. This time, for a change, it's set on the `Parent` object's prototype `p`, not on the `Parent` constructor. This is why `toString()` uses it as `this.uber`, as opposed to `this.constructor.uber`. This is just an illustration that you can shape your favorite inheritance pattern in any way you see fit. Let's test it out:

```
>td.toString();
"Shape, Shape"
```

`TwoDShape` didn't redefine the `name` property, hence the repetition. It can do that at any time and (the prototype chain being live) all the instances "see" the update:

```
>TwoDShape.prototype.name = "2D shape";
>td.toString();
"Shape, 2D shape"
```

Heads-up when copying by reference

The fact that objects (including functions and arrays) are copied by reference could sometimes lead to results you don't expect.

Let's create two constructor functions and add properties to the prototype of the first one:

```
> function Papa() {}
>function Wee() {}
>Papa.prototype.name = 'Bear';
>Papa.prototype.owns = ["porridge", "chair", "bed"];
```

Now, let's have `Wee` inherit from `Papa` (either `extend()` or `extend2()` will do):

```
>extend2(Wee, Papa);
```

Using `extend2()`, the `Wee` function's prototype inherited the properties of `Papa.prototype` as its own.

```
>Wee.prototype.hasOwnProperty('name');
true

>Wee.prototype.hasOwnProperty('owns');
true
```

The `name` property is primitive so a new copy of it is created. The property `owns` is an array object so it's copied by reference:

```
>Wee.prototype.owns;
["porridge", "chair", "bed"]
```

```
>Wee.prototype.owns=== Papa.prototype.owns;
```
true

Changing the `Wee` function's copy of `name` doesn't affect `Papa`:

```
>Wee.prototype.name += ', Little Bear';
```
"Bear, Little Bear"

```
>Papa.prototype.name;
```
"Bear"

Changing the `Wee` function's `owns` property, however, affects `Papa`, because both properties point to the same array in memory.

```
>Wee.prototype.owns.pop();
```
"bed"

```
>Papa.prototype.owns;
```
["porridge", "chair"]

It's a different story when you completely overwrite the `Wee` function's copy of `owns` with another object (as opposed to modifying the existing one). In this case `Papa.owns` keeps pointing to the old object, while `Wee.owns` points to a new one.

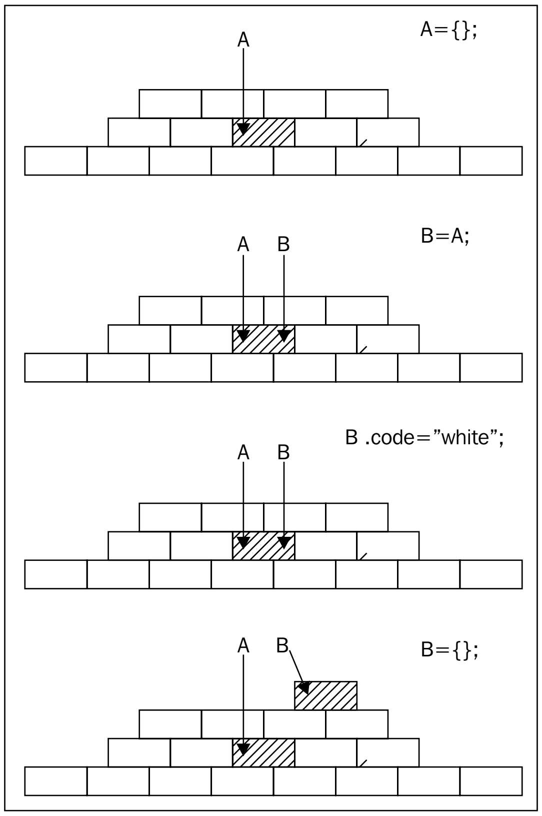
```
>Wee.prototype.owns= ["empty bowl", "broken chair"];
>Papa.prototype.owns.push('bed');
>Papa.prototype.owns;
```
["porridge", "chair", "bed"]

Think of an object as something that is created and stored in a physical location in memory. Variables and properties merely point to this location, so when you assign a brand new object to `Wee.prototype.owns` you essentially say, "Hey, forget about this other old object, move your pointer to this new one instead".

The following diagram illustrates what happens if you imagine the memory being a heap of objects (like a wall of bricks) and you point to (refer to) some of these objects.

- A new object is created and A points to it.
- A new variable B is created and made equal to A, meaning it now points to the same place where A is pointing to.
- A property color is changed using the B handle (pointer). The brick is now white. A check for `A.color === "white"` would be true.

- A new object is created and the B variable/pointer is recycled to point to that new object. A and B are now pointing to different parts of the memory pile, they have nothing in common and changes to one of them don't affect the other:



If you want to address the problem that objects are copied by reference, consider a deep copy, described further.

Objects inherit from objects

All of the examples so far in this chapter assume that you create your objects with constructor functions and you want objects created with one constructor to inherit properties that come from another constructor. However, you can also create objects without the help of a constructor function, just by using the object literal and this is, in fact, less typing. So how about inheriting those?

In Java or PHP, you define classes and have them inherit from other classes. That's why you'll see the term *classical*, because the OO functionality comes from the use of classes. In JavaScript, there are no classes, so programmers that come from a classical background resort to constructor functions because constructors are the closest to what they are used to. In addition, JavaScript provides the `new` operator, which can further suggest that JavaScript is like Java. The truth is that, in the end, it all comes down to objects. The first example in this chapter used this syntax:

```
Child.prototype = new Parent();
```

Here, the `Child` constructor (or class, if you will) inherits from `Parent`. But this is done through creating an object using `new Parent()` and inheriting from it. That's why this is also referred to as a *pseudo-classical inheritance pattern*, because it resembles classical inheritance, although it isn't (no classes are involved).

So why not get rid of the middleman (the constructor/class) and just have objects inherit from objects? In `extend2()` the properties of the parent prototype object were copied as properties of the child prototype object. The two prototypes are in essence just objects. Forgetting about prototypes and constructor functions, you can simply take an object and copy all of its properties into another object.

You already know that objects can start as a "blank canvas" without any own properties by using `var o = {};` and then get properties later. But, instead of starting fresh, you can start by copying all of the properties of an existing object. Here's a function that does exactly that: it takes an object and returns a new copy of it.

```
function extendCopy(p) {
var c = {};
for (vari in p) {
c[i] = p[i];
}
c.uber = p;
return c;
}
```

Simply copying all of the properties is a straightforward pattern, and it's widely used. Let's see this function in action. You start by having a base object:

```
var shape = {
name: 'Shape',
toString: function () {
return this.name;
}
};
```

In order to create a new object that builds upon the old one, you can call the function `extendCopy()` which returns a new object. Then, you can augment the new object with additional functionality.

```
vartwoDee = extendCopy(shape);
twoDee.name = '2D shape';
twoDee.toString = function () {
return this.uber.toString() + ', ' + this.name;
};
```

A triangle object that inherits the 2D shape object:

```
var triangle = extendCopy(twoDee);
triangle.name = 'Triangle';
triangle.getArea = function () {
return this.side * this.height / 2;
};
```

Using the triangle:

```
>triangle.side = 5;
>triangle.height = 10;
>triangle.getArea();
25

>triangle.toString();
"Shape, 2D shape, Triangle"
```

A possible drawback of this method is the somewhat verbose way of initializing the new `triangle` object, where you manually set values for `side` and `height`, as opposed to passing them as values to a constructor. But, this is easily resolved by having a function, for example, called `init()` (or `__construct()` if you come from PHP) that acts as a constructor and accepts initialization parameters. Or, have `extendCopy()` accept two parameters: an object to inherit from and another object literal of properties to add to the copy before it's returned, in other words just merge two objects.

Deep copy

The function `extendCopy()`, discussed previously, creates what is called a shallow copy of an object, just like `extend2()` before that. The opposite of a shallow copy would be, naturally, a deep copy. As discussed previously (in the *Heads-up when copying by reference* section), when you copy objects you only copy pointers to the location in memory where the object is stored. This is what happens in a shallow copy. If you modify an object in the copy, you also modify the original. The deep copy avoids this problem.

The deep copy is implemented in the same way as the shallow copy: you loop through the properties and copy them one by one. But, when you encounter a property that points to an object, you call the deep copy function again:

```
function deepCopy(p, c) {
  c = c || {};
  for (vari in p) {
    if (p.hasOwnProperty(i)) {
      if (typeof p[i] === 'object') {
        c[i] = Array.isArray(p[i]) ? [] : {};
deepCopy(p[i], c[i]);
      } else {
        c[i] = p[i];
      }
    }
  }
  return c;
}
```

Let's create an object that has arrays and a sub-object as properties.

```
var parent = {
numbers: [1, 2, 3],
letters: ['a', 'b', 'c'],
obj: {
prop: 1
},
bool: true
};
```

Let's test this by creating a deep copy and a shallow copy. Unlike the shallow copy, when you update the `numbers` property of a deep copy, the original is not affected.

```
>varmydeep = deepCopy(parent);
>varmyshallow = extendCopy(parent);
>mydeep.numbers.push(4,5,6);
6

>mydeep.numbers;
[1, 2, 3, 4, 5, 6]

>parent.numbers;
[1, 2, 3]

>myshallow.numbers.push(10);
4
```

```
>myshallow.numbers;
[1, 2, 3, 10]

>parent.numbers;
[1, 2, 3, 10]

>mydeep.numbers;
[1, 2, 3, 4, 5, 6]
```

Two side notes about the `deepCopy()` function:

- Filtering out non-own properties with `hasOwnProperty()` is always a good idea to make sure you don't carry over someone's additions to the core prototypes.
- `Array.isArray()` exists since ES5 because it's surprisingly hard otherwise to tell real arrays from objects. The best cross-browser solution (if you need to define `isArray()` in ES3 browsers) looks a little hacky, but it works:

```
if (Array.isArray !== "function") {
Array.isArray = function (candidate) {
    return
Object.prototype.toString.call(candidate) ===
'[object Array]';
};
}
```

object()

Based on the idea that objects inherit from objects, Douglas Crockford advocates the use of an `object()` function that accepts an object and returns a new one that has the parent as a prototype.

```
function object(o) {
function F() {}
F.prototype = o;
return new F();
}
```

If you need access to an `uber` property, you can modify the `object()` function like so:

```
function object(o) {
var n;
function F() {}
F.prototype = o;
n = new F();
n.uber = o;
return n;
}
```

Using this function is the same as using the `extendCopy()`: you take an object such as `twoDee`, create a new object from it and then proceed to augmenting the new object.

```
var triangle = object(twoDee);
triangle.name = 'Triangle';
triangle.getArea = function () {
return this.side * this.height / 2;
};
```

The new triangle still behaves the same way:

```
>triangle.toString();
"Shape, 2D shape, Triangle"
```

This pattern is also referred to as **prototypal inheritance**, because you use a parent object as the prototype of a child object. It's also adopted and built upon in ES5 and called `Object.create()`. For example:

```
>var square = Object.create(triangle);
```

Using a mix of prototypal inheritance and copying properties

When you use inheritance, you will most likely want to take already existing functionality and then build upon it. This means creating a new object by inheriting from an existing object and then adding additional methods and properties. You can do this with one function call, using a combination of the last two approaches just discussed.

You can:

- Use prototypal inheritance to use an existing object as a prototype of a new one
- Copy all of the properties of another object into the newly created one

```
function objectPlus(o, stuff) {
var n;
function F() {}
F.prototype = o;
n = new F();
n.uber = o;

for (vari in stuff) {
n[i] = stuff[i];
}
return n;
}
```

This function takes an object `o` to inherit from and another object `stuff` that has the additional methods and properties that are to be copied. Let's see this in action.

Start with the base `shape` object:

```
var shape = {
name: 'Shape',
toString: function () {
return this.name;
}
};
```

Create a 2D object by inheriting `shape` and adding more properties. The additional properties are simply created with an object literal.

```
vartwoDee = objectPlus(shape, {
name: '2D shape',
toString: function () {
return this.uber.toString() + ', ' + this.name;
}
});
```

Now, let's create a `triangle` object that inherits from 2D and adds more properties.

```
var triangle = objectPlus(twoDee, {
name: 'Triangle',
getArea: function () {
return this.side * this.height / 2;
},
side: 0,
height: 0
});
```

Testing how it all works by creating a concrete triangle `my` with defined `side` and `height`:

```
var my = objectPlus(triangle, {
side: 4, height: 4
});
>my.getArea();
8
```

```
>my.toString();
"Shape, 2D shape, Triangle, Triangle"
```

The difference here, when executing `toString()`, is that the **Triangle** name is repeated twice. That's because the concrete instance was created by inheriting `triangle`, so there was one more level of inheritance. You could give the new instance a name:

```
>objectPlus(triangle, {
side: 4,
height: 4,
 name: 'My 4x4'
}).toString();
"Shape, 2D shape, Triangle, My 4x4"
```

This `objectPlus()` is even closer to ES5's `Object.create()` only the ES5 one takes the additional properties (the second argument) using something called property descriptors (discussed in *Chapter 11*, *Built-in Objects*).

Multiple inheritance

Multiple inheritance is where a child inherits from more than one parent. Some OO languages support multiple inheritance out of the box, and some don't. You can argue both ways: that multiple inheritance is convenient, or that it's unnecessary, complicates application design, and it's better to use an inheritance chain instead. Leaving the discussion of multiple inheritance's pros and cons for the long, cold winter nights, let's see how you can do it in practice in JavaScript.

The implementation can be as simple as taking the idea of inheritance by copying properties, and expanding it so that it takes an unlimited number of input objects to inherit from.

Let's create a `multi()` function that accepts any number of input objects. You can wrap the loop that copies properties in another loop that goes through all the objects passed as `arguments` to the function.

```
function multi() {
var n = {}, stuff, j = 0, len = arguments.length;
for (j = 0; j <len; j++) {
stuff = arguments[j];
for (vari in stuff) {
      if (stuff.hasOwnProperty(i)) {
n[i] = stuff[i];
      }
}
}
return n;
}
```

Let's test this by creating three objects: `shape`, `twoDee`, and a third, unnamed object. Then, creating a `triangle` object means calling `multi()` and passing all three objects.

```
var shape = {
name: 'Shape',
toString: function () {
return this.name;
}
};

vartwoDee = {
name: '2D shape',
dimensions: 2
};

var triangle = multi(shape, twoDee, {
name: 'Triangle',
getArea: function () {
return this.side * this.height / 2;
},
side: 5,
height: 10
});
```

Does this work? Let's see. The method `getArea()` should be an own property, `dimensions` should come from `twoDee` and `toString()` from `shape`.

```
>triangle.getArea();
25

>triangle.dimensions;
2

>triangle.toString();
"Triangle"
```

Bear in mind that `multi()` loops through the input objects in the order they appear and if it happens that two of them have the same property, the last one wins.

Mixins

You might come across the term *mixin*. Think of a mixin as an object that provides some useful functionality but is not meant to be inherited and extended by sub-objects. The approach to multiple inheritance outlined previously can be considered an implementation of the mixins idea. When you create a new object you can pick and choose any other objects to mix into your new object. By passing them all to `multi()` you get all their functionality without making them part of the inheritance tree.

Parasitic inheritance

If you like the fact that you can have all kinds of different ways to implement inheritance in JavaScript, and you're hungry for more, here's another one. This pattern, courtesy of Douglas Crockford, is called parasitic inheritance. It's about a function that creates objects by taking all of the functionality from another object into a new one, augmenting the new object, and returning it, "pretending that it has done all the work".

Here's an ordinary object, defined with an object literal, and unaware of the fact that it's soon going to fall victim to parasitism:

```
vartwoD = {
name: '2D shape',
dimensions: 2
};
```

A function that creates triangle objects could:

- Use `twoD` object as a prototype of an object called `that` (similar to `this` for convenience). This can be done in any way you saw previously, for example using the `object()` function or copying all the properties.
- Augment `that` with more properties.
- Return `that`.

```
function triangle(s, h) {
var that = object(twoD);
that.name ='Triangle';
that.getArea = function () {
return this.side * this.height / 2;
};
that.side = s;
that.height = h;
return that;
}
```

Because `triangle()` is a normal function, not a constructor, it doesn't require the `new` operator. But because it returns an object, calling it with `new` by mistake works too.

```
>var t = triangle(5, 10);
>t.dimensions;
2

>vart2 = new triangle(5,5);
>t2.getArea();
12.5
```

Note, that `that` is just a name; it doesn't have a special meaning, the way `this` does.

Borrowing a constructor

One more way of implementing inheritance (the last one in the chapter, I promise) has to do again with constructor functions, and not the objects directly. In this pattern the constructor of the child calls the constructor of the parent using either `call()` or `apply()` methods. This can be called *stealing a constructor*, or *inheritance by borrowing a constructor* if you want to be more subtle about it.

`call()` and `apply()` were discussed in *Chapter 4, Objects* but here's a refresher: they allow you to call a function and pass an object that the function should bind to its `this` value. So for inheritance purposes, the child constructor calls the parent's constructor and binds the child's newly-created `this` object as the parent's `this`.

Let's have this parent constructor `Shape()`:

```
function Shape(id) {
this.id = id;
}
Shape.prototype.name = 'Shape';
Shape.prototype.toString = function () {
return this.name;
};
```

Now, let's define `Triangle()` which uses `apply()` to call the `Shape()` constructor, passing `this` (an instance created with `new Triangle()`) and any additional arguments.

```
function Triangle() {
Shape.apply(this, arguments);
}
Triangle.prototype.name = 'Triangle';
```

Note, that both `Triangle()` and `Shape()` have added some extra properties to their prototypes.

Now, let's test this by creating a new triangle object:

```
>var t = new Triangle(101);
>t.name;
"Triangle"
```

The new triangle object inherits the `id` property from the parent, but it doesn't inherit anything added to the parent's prototype:

```
>t.id;
101
```

```
>t.toString();
"[object Object]"
```

The triangle failed to get the `Shape` function's prototype properties because there was never a `new Shape()` instance created, so the prototype was never used. But, you saw how to do this at the beginning of this chapter. You can redefine `Triangle` like this:

```
function Triangle() {
Shape.apply(this, arguments);
}
Triangle.prototype = new Shape();
Triangle.prototype.name = 'Triangle';
```

In this inheritance pattern, the parent's own properties are recreated as the child's own properties. If a child inherits an array or other object, it's a completely new value (not a reference) and modifying it won't affect the parent.

The drawback is that the parent's constructor gets called twice: once with `apply()` to inherit own properties and once with `new` to inherit the prototype. In fact the own properties of the parent are inherited twice. Let's take this simplified scenario:

```
function Shape(id) {
this.id = id;
}
function Triangle() {
Shape.apply(this, arguments);
}
Triangle.prototype = new Shape(101);
```

Creating a new instance:

```
>var t = new Triangle(202);
>t.id;
202
```

There's an own property `id`, but there's also one that comes down the prototype chain, ready to shine through:

```
>t.__proto__.id;
101

> delete t.id;
true

>t.id;
101
```

Borrow a constructor and copy its prototype

The problem of the double work performed by calling the constructor twice can easily be corrected. You can call `apply()` on the parent constructor to get all own properties and then copy the prototype's properties using a simple iteration (or `extend2()` as discussed previously).

```
function Shape(id) {
this.id = id;
}
Shape.prototype.name = 'Shape';
Shape.prototype.toString = function () {
return this.name;
};

function Triangle() {
Shape.apply(this, arguments);
}
extend2(Triangle, Shape);
Triangle.prototype.name = 'Triangle';
```

Testing:

```
>var t = new Triangle(101);
>t.toString();
"Triangle"
>t.id;
101
```

No double inheritance:

```
>typeoft.__proto__.id;
"undefined"
```

`extend2()` also gives access to `uber` if needed:

```
>t.uber.name;
"Shape"
```

Summary

In this chapter you learned quite a few ways (patterns) for implementing inheritance and the following table summarizes them. The different types can roughly be divided into:

- Patterns that work with constructors
- Patterns that work with objects

You can also classify the patterns based on whether they:

- Use the prototype
- Copy properties
- Do both (copy properties of the prototype)

#	Name	Example	Classification	Notes
1	Prototype chaining (pseudo-classical)	`Child.prototype = new Parent();`	• Works with constructors • Uses the prototype chain	• The default mechanism. • Tip: move all properties/ methods that are meant to be reused to the prototype, add the non-reusable as own properties.

#	Name	Example	Classification	Notes

2	Inherit only the prototype	`Child.prototype =`
		`Parent.prototype;`
			• Works with constructors
			• Copies the prototype (no prototype chain, all share the same prototype object)
				• More efficient, no new instances are created just for the sake of inheritance.
				• Prototype chain lookup during runtime- is fast, since there's no chain.
				• Drawback: children can modify parents' functionality.

3	Temporary constructor	`function`
		`extend(Child,`
		`Parent) {`
		`  var F =`
		`function(){};`
		`  F.prototype =`
		`Parent.prototype;`
		`  Child.prototype`
		`= new F();`
		`  Child.prototype.`
		`constructor =`
		`Child;`
		`  Child.uber =`
		`Parent.prototype;`
		`}`
			• Works with constructors
			• Uses the prototype chain
				• Unlike #1, it only inherits properties of the prototype. Own properties (created with this inside the constructor) are not inherited.
				• Provides convenient access to the parent (through uber).

4	Copying the prototype properties	`function`
		`extend2(Child,`
		`Parent) {`
		`var p = Parent.`
		`prototype;`
		`var c = Child.`
		`prototype;`
		` for (vari in p) {`
		` c[i] = p[i];`
		`  }`
		`c.uber = p;`
		`}`
			• Works with constructors
			• Copies properties
			• Uses the prototype chain
				• All properties of the parent prototype become properties of the child prototype
				• No need to create a new object only for inheritance purposes
				• Shorter prototype chains

#	Name	Example	Classification	Notes

5	Copy all properties
	(shallow copy)
		`function extendCopy(p) {`
		`var c = {};`
		` for (vari in p) {`
		` c[i] = p[i];`
		` }`
		`c.uber = p;`
		` return c;`
		`}`
			• Works with objects
			• Copies properties
				• Simple
				• Doesn't use prototypes

6	Deep copy	Same as above, but recurse into objects	• Works with objects • Copies properties	Same as #5 but clones objects and arrays

7	Prototypal inheritance	`function object(o)`
		`{`
		` function F() {}`
		`F.prototype = o;`
		` return new F();`
		`}`
			• Works with objects
			• Uses the prototype chain
				• No pseudo-classes, objects inherit from objects
				• Leverages the benefits of the prototype

8	Extend and augment	`function objectPlus(o, stuff) {`
		`var n;`
		` function F() {}`
		`F.prototype = o;`
		` n = new F();`
		`n.uber = o;`
		` for (vari in stuff) {`
		` n[i] = stuff[i];`
		` }`
		` return n;`
		`}`
			• Works with objects
			• Uses the prototype chain
			• Copies properties
				• Mix of prototypal inheritance (#7) and copying properties (#5)
				• One function call to inherit and extend at the same time

#	Name	Example	Classification	Notes
9	Multiple inheritance	`function multi() { var n = {}, stuff, j = 0, len = arguments.length; for (j = 0; j <len; j++) { stuff = arguments[j]; for (vari in stuff) { n[i] = stuff[i]; } } return n; }`	• Works with objects • Copies properties	• A mixin-style implementation • Copies all the properties of all the parent objects in the order of appearance
10	Parasitic inheritance	`function parasite(victim) { var that = object(victim); that.more = 1; return that; }`	• Works with objects • Uses the prototype chain	• Constructor-like function creates objects • Copies an object, augments and returns the copy
11	Borrowing constructors	`function Child() { Parent.apply(this, arguments); }`	Works with constructors	• Inherits only own properties • Can be combined with #1 to inherit the prototype too • Convenient way to deal with the issues when a child inherits a property that is an object (and therefore passed by reference)

12	Borrow a constructor and copy the prototype	`function Child() { Parent.apply(this, arguments); }`
		
		`extend2(Child, Parent);`
			• Works with constructors
			• Uses the prototype chain
			• Copies properties
				• Combination of #11 and #4
				• Allows you to inherit both own properties and prototype properties without calling the parent constructor twice

Given so many options, you must be wondering: which is the right one? That depends on your style and preferences, your project, task, and team. Are you more comfortable thinking in terms of classes? Then pick one of the methods that work with constructors. Are you going to need just one or a few instances of your "class"? Then choose an object-based pattern.

Are these the only ways of implementing inheritance? No. You can chose a pattern from the preceding table or you can mix them, or you can think of your own. The important thing is to understand and be comfortable with objects, prototypes, and constructors; the rest is just pure joy.

Case study – drawing shapes

Let's finish off this chapter with a more practical example of using inheritance. The task is to be able to calculate the area and the perimeter of different shapes, as well as to draw them, while reusing as much code as possible.

Analysis

Let's have one `Shape` constructor that contains all of the common parts. From there, let's have `Triangle`, `Rectangle`, and `Square` constructors, all inheriting from `Shape`. A square is really a rectangle with the same-length sides, so let's reuse `Rectangle` when building the `Square`.

In order to define a shape, you'll need points with `x` and `y` coordinates. A generic shape can have any number of points. A triangle is defined with three points, a rectangle (to keep it simpler)—with one point and the lengths of the sides. The perimeter of any shape is the sum of its sides' lengths. Calculating the area is shape-specific and will be implemented by each shape.

The common functionality in `Shape` would be:

- A `draw()` method that can draw any shape given the points
- A `getParameter()` method
- A property that contains an array of `points`
- Other methods and properties as needed

For the drawing part let's use a `<canvas>` tag. It's not supported in early IEs, but hey, this is just an exercise.

Let's have two other helper constructors—`Point` and `Line`. `Point` will help when defining shapes; `Line` will make calculations easier, as it can give the length of the line connecting any two given points.

You can play with a working example here: `http://www.phpied.com/files/canvas/`. Just open your console and start creating new shapes as you'll see in a moment.

Implementation

Let's start by adding a canvas tag to a blank HTML page:

```
<canvas height="600" width="800" id="canvas" />
```

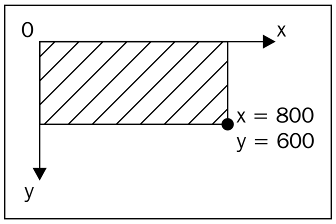
Then, put the JavaScript code inside `<script>` tags:

```
<script>
// ... code goes here
</script>
```

Now, let's take a look at what's in the JavaScript part. First, the helper `Point` constructor. It just can't get any more trivial than this:

```
function Point(x, y) {
this.x = x;
this.y = y;
}
```

Bear in mind that the coordinates of the points on the canvas start from x=0, y=0, which is the top left. The bottom right will be x = 800, y = 600:



Next, the `Line` constructor. It takes two points and calculates the length of the line between them, using the Pythagorean Theorem a2 + b2 = c2 (imagine a right-angled triangle where the hypotenuse connects the two given points).

```
function Line(p1, p2) {
this.p1 = p1;
this.p2 = p2;
this.length = Math.sqrt(
Math.pow(p1.x - p2.x, 2) +
Math.pow(p1.y - p2.y, 2)
);
}
```

Next, comes the `Shape` constructor. The shapes will have their points (and the lines that connect them) as own properties. The constructor also invokes an initialization method, `init()`, that will be defined in the prototype.

```
function Shape() {
this.points = [];
this.lines= [];
this.init();
}
```

Now the big part: the methods of `Shape.prototype`. Let's define all of these methods using the object literal notation. Refer to the comments for guidelines as to what each method does.

```
Shape.prototype = {
  // reset pointer to constructor
  constructor: Shape,

  // initialization, sets this.context to point
  // to the context if the canvas object
init: function () {
    if (this.context === undefined) {
var canvas = document.getElementById('canvas');
Shape.prototype.context = canvas.getContext('2d');
    }
  },

  // method that draws a shape by looping through this.points
  draw: function () {
vari, ctx = this.context;
ctx.strokeStyle = this.getColor();
ctx.beginPath();
ctx.moveTo(this.points[0].x, this.points[0].y);
    for (i = 1; i<this.points.length; i++) {
ctx.lineTo(this.points[i].x, this.points[i].y);
    }
ctx.closePath();
ctx.stroke();
  },

  // method that generates a random color
getColor: function () {
vari, rgb = [];
    for (i = 0; i< 3; i++) {
rgb[i] = Math.round(255 * Math.random());
```

```
      }
      return 'rgb(' + rgb.join(',') + ')';
    },

    // method that loops through the points array,
    // creates Line instances and adds them to this.lines
  getLines: function () {
      if (this.lines.length> 0) {
        return this.lines;
      }
  vari, lines = [];
      for (i = 0; i<this.points.length; i++) {
        lines[i] = new Line(this.points[i],
  this.points[i + 1] || this.points[0]);
      }
  this.lines = lines;
      return lines;
    },

    // shell method, to be implemented by children
  getArea: function () {},

    // sums the lengths of all lines
  getPerimeter: function () {
  vari, perim = 0, lines = this.getLines();
      for (i = 0; i<lines.length; i++) {
  perim += lines[i].length;
      }
      return perim;
    }
  };
```

Now, the children constructor functions. `Triangle` first:

```
  function Triangle(a, b, c) {
  this.points = [a, b, c];

  this.getArea = function () {
  var p = this.getPerimeter(),
          s = p / 2;
      return Math.sqrt(
        s
        * (s - this.lines[0].length)
        * (s - this.lines[1].length)
        * (s - this.lines[2].length));
    };
  }
```

The `Triangle` constructor takes three point objects and assigns them to `this.points` (its own collection of points). Then it implements the `getArea()` method, using Heron's formula:

```
Area = s(s-a)(s-b)(s-c)
```

`s` is the semi-perimeter (perimeter divided by two).

Next, comes the `Rectangle` constructor. It receives one point (the upper-left point) and the lengths of the two sides. Then, it populates its `points` array starting from that one point.

```
function Rectangle(p, side_a, side_b){
this.points = [
p,
new Point(p.x + side_a, p.y),// top right
new Point(p.x + side_a, p.y + side_b), // bottom right
new Point(p.x, p.y + side_b)// bottom left
];
this.getArea = function () {
return side_a * side_b;
};
}
```

The last child constructor is `Square`. A square is a special case of a rectangle, so it makes sense to reuse `Rectangle`. The easiest thing to do here is to borrow the constructor.

```
function Square(p, side){
Rectangle.call(this, p, side, side);
}
```

Now that all constructors are done, let's take care of inheritance. Any pseudo-classical pattern (one that works with constructors as opposed to objects) will do. Let's try using a modified and simplified version of the prototype-chaining pattern (the first method described in this chapter). This pattern calls for creating a new instance of the parent and setting it as the child's prototype. In this case, it's not necessary to have a new instance for each child—they can all share it.

```
(function () {
var s = new Shape();
Triangle.prototype = s;
Rectangle.prototype = s;
Square.prototype = s;
})();
```

Testing

Let's test this by drawing shapes. First, define three points for a triangle:

```
>varp1 = new Point(100, 100);
>varp2 = new Point(300, 100);
>varp3 = new Point(200, 0);
```

Now, you can create a triangle by passing the three points to the `Triangle` constructor:

```
>var t = new Triangle(p1, p2, p3);
```

You can call the methods to draw the triangle on the canvas and get its area and perimeter:

```
>t.draw();
>t.getPerimeter();
482.842712474619

>t.getArea();
10000.000000000002
```

Now, let's play with a rectangle instance:

```
>var r = new Rectangle(new Point(200, 200), 50, 100);
>r.draw();
>r.getArea();
5000

>r.getPerimeter();
300
```

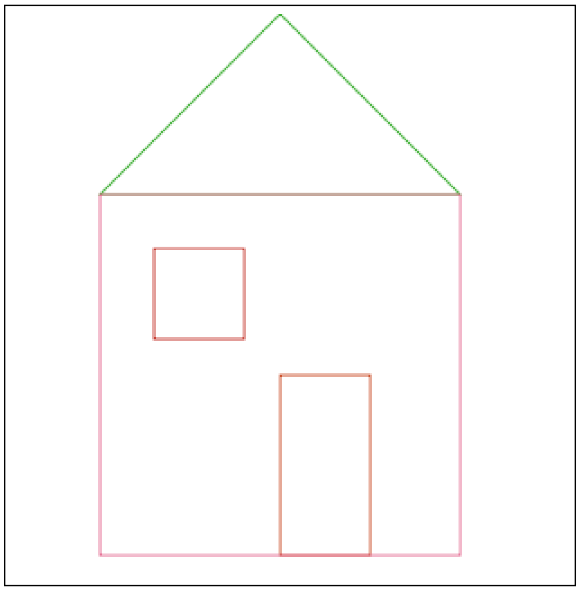
And finally, a square:

```
>var s = new Square(new Point(130, 130), 50);
>s.draw();
>s.getArea();
2500

>s.getPerimeter();
200
```

It's fun to draw these shapes. You can also be as lazy as the following example, which draws another square, reusing a triangle's point:

```
> new Square(p1, 200).draw();
```

The result of the tests will be something like this:



Exercises

1. Implement multiple inheritance but with a prototypal inheritance pattern, not property copying. For example:

   ```
   var my = objectMulti(obj, another_obj, a_third, {
   additional: "properties"
   });
   ```

   The property `additional` should be an own property, all the rest should be mixed into the prototype.

2. Use the canvas example to practice. Try out different things, for example:
    - Draw a few triangles, squares, and rectangles.
    - Add constructors for more shapes, such as `Trapezoid`, `Rhombus`, `Kite`, and `Pentagon`. If you want to learn more about the canvas tag, create a `Circle` constructor too. It will need to overwrite the `draw()` method of the parent.
    - Can you think of another way to approach the problem and use another type of inheritance?
    - Pick one of the methods that uses `uber` as a way for a child to access its parent. Add functionality where the parents can keep track of who their children are. Perhaps by using a property that contains a `children` array?

7
The Browser Environment

You know that JavaScript programs need a host environment. Most of what you learned so far in this book was related to core ECMAScript/JavaScript and can be used in many different host environments. Now, let's shift the focus to the browser, since this is the most popular and natural host environment for JavaScript programs. In this chapter, you will learn about the following elements:

- The **Browser Object Model** (**BOM**)
- The **Document Object Model** (**DOM**)
- Browser events
- The `XMLHttpRequest` object

Including JavaScript in an HTML page

To include JavaScript in an HTML page, you need to use the `<script>` tag as follows:

```
<!DOCTYPE>
<html>
  <head>
    <title>JS test</title>
    <script src="somefile.js"></script>
  </head>
  <body>
    <script>
      var a = 1;
      a++;
    </script>
  </body>
</html>
```

In this example, the first `<script>` tag includes an external file, `somefile.js`, which contains JavaScript code. The second `<script>` tag includes the JavaScript code directly in the HTML code of the page. The browser executes the JavaScript code in the sequence it finds it on the page and all the code in all tags share the same global namespace. This means that when you define a variable in `somefile.js`, it also exists in the second `<script>` block.

BOM and DOM – an overview

The JavaScript code in a page has access to a number of objects. These objects can be divided into the following types:

- **Core ECMAScript objects**: All the objects mentioned in the previous chapters
- **DOM**: Objects that have to do with the currently loaded page (the page is also called the document)
- **BOM**: Objects that deal with everything outside the page (the browser window and the desktop screen)

DOM stands for Document Object Model and BOM for Browser Object Model.

The DOM is a standard, governed by the **World Wide Web Consortium** (**W3C**) and has different versions, called levels, such as DOM Level 1, DOM Level 2, and so on. Browsers in use today have different degrees of compliance with the standard but in general, they almost all completely implement DOM Level 1. The DOM was standardized post-factum, after the browser vendors had each implemented their own ways to access the document. The legacy part (from before the W3C took over) is still around and is referred to as DOM 0, although no real DOM Level 0 standard exists. Some parts of DOM 0 have become de-facto standards as all major browsers support them. Some of these were added to the DOM Level 1 standard. The rest of DOM 0 that didn't find its way to DOM 1 is too browser-specific and won't be discussed here.

BOM historically has not been a part of any standard. Similar to DOM 0, it has a subset of objects that is supported by all major browsers, and another subset that is browser-specific. The HTML5 standard codifies common behavior among browsers, and it includes common BOM objects. Additionally, mobile devices come with their specific objects (and HTML5 aims to standardize those as well) which traditionally have not been necessary for desktop computers, but make sense in a mobile world, for example, geolocation, camera access, vibration, touch events, telephony, and SMS.

This chapter discusses only cross-browser subsets of BOM and DOM Level 1 (unless noted otherwise in the text). Even these safe subsets constitute a large topic, and a full reference is beyond the scope of this book. You can also consult the following references:

- Mozilla DOM reference (`http://developer.mozilla.org/en/docs/Gecko_DOM_Reference`)
- Mozilla's HTML5 wiki (`https://developer.mozilla.org/en-US/docs/HTML/HTML5`)
- Microsoft's documentation for Internet Explorer (`http://msdn2.microsoft.com/en-us/library/ms533050(vs.85).aspx`)
- W3C's DOM specifications (`http://www.w3.org/DOM/DOMTR`)

BOM

The Browser Object Model (BOM) is a collection of objects that give you access to the browser and the computer screen. These objects are accessible through the global object `window`.

The window object revisited

As you know already, in JavaScript there's a global object provided by the host environment. In the browser environment, this global object is accessible using `window`. All global variables are also accessible as properties of the `window` object as follows:

```
> window.somevar = 1;
  1

> somevar;
  1
```

Also, all of the core JavaScript functions (discussed in *Chapter 2*, *Primitive Data Types, Arrays, Loops, and Conditions*) are methods of the global object. Have a look at the following code snippet:

```
> parseInt('123a456');
  123

> window.parseInt('123a456');
  123
```

In addition to being a reference to the global object, the `window` object also serves a second purpose providing information about the browser environment. There's a `window` object for every frame, iframe, pop up, or browser tab.

Let's see some of the browser-related properties of the `window` object. Again, these can vary from one browser to another, so let's only consider the properties that are implemented consistently and reliably across all major browsers.

window.navigator

The `navigator` is an object that has some information about the browser and its capabilities. One property is `navigator.userAgent`, which is a long string of browser identification. In Firefox, you'll get the following output:

```
> window.navigator.userAgent;
```

"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_3) AppleWebKit/536.28.10 (KHTML, like Gecko) Version/6.0.3 Safari/536.28.10"

The `userAgent` string in Microsoft Internet Explorer would be something like the following:

"Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; Trident/6.0)"

Because the browsers have different capabilities, developers have been using the `userAgent` string to identify the browser and provide different versions of the code. For example, the following code searches for the presence of the string `MSIE` to identify Internet Explorer:

```
if (navigator.userAgent.indexOf('MSIE') !== -1) {
  // this is IE
} else {
  // not IE
}
```

It's better not to rely on the user agent string, but to use feature sniffing (also called capability detection) instead. The reason for this is that it's hard to keep track of all browsers and their different versions. It's much easier to simply check if the feature you intend to use is indeed available in the user's browser. For example have a look at the following code:

```
if (typeof window.addEventListener === 'function') {
  // feature is supported, let's use it
} else {
  // hmm, this feature is not supported, will have to
  // think of another way
}
```

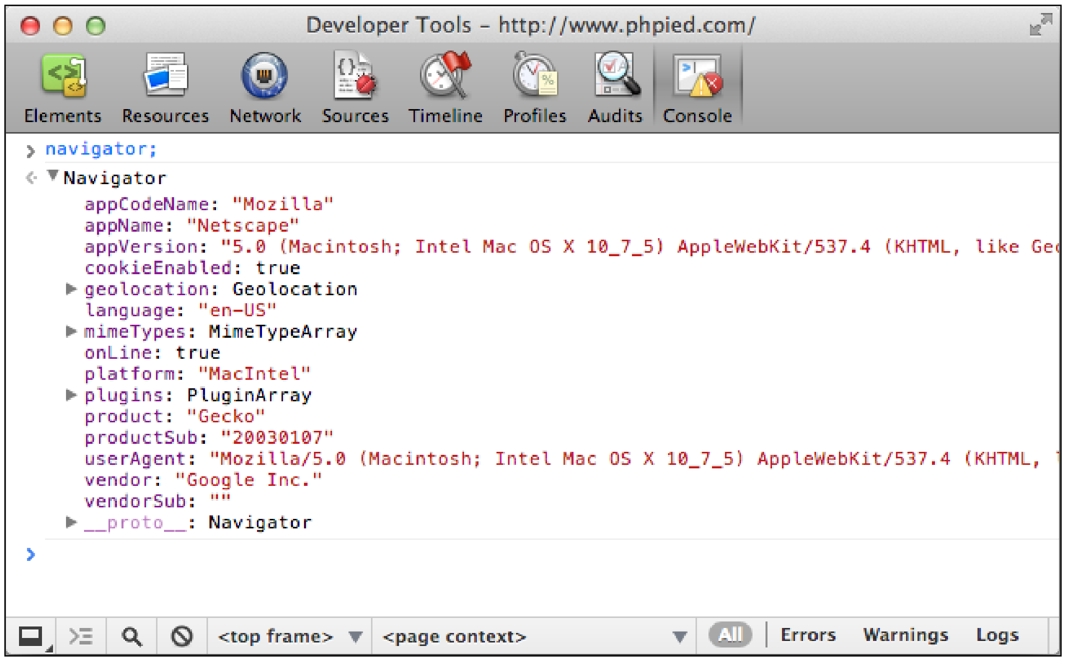
Another reason to avoid user agent sniffing is that some browsers allow users to modify the string and pretend they are using a different browser.

Your console is a cheat sheet

The console lets you inspect what's in an object and this includes all the BOM and DOM properties. Just type the following code:

```
> navigator;
```

Then click on the result. The result is a list of properties and their values, as shown in the following screenshot:



```
> navigator;
<· ▼Navigator
     appCodeName: "Mozilla"
     appName: "Netscape"
     appVersion: "5.0 (Macintosh; Intel Mac OS X 10_7_5) AppleWebKit/537.4 (KHTML, like Ge
     cookieEnabled: true
   ▶ geolocation: Geolocation
     language: "en-US"
   ▶ mimeTypes: MimeTypeArray
     onLine: true
     platform: "MacIntel"
   ▶ plugins: PluginArray
     product: "Gecko"
     productSub: "20030107"
     userAgent: "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_5) AppleWebKit/537.4 (KHTML,
     vendor: "Google Inc."
     vendorSub: ""
   ▶ __proto__: Navigator
>
```

window.location

The `location` property points to an object that contains information about the URL of the currently loaded page. For example, `location.href` is the full URL and `location.hostname` is only the domain. With a simple loop, you can see the full list of properties of the `location` object.

Imagine you're on a page with a URL `http://search.phpied.com:8080/search?q=java&what=script#results`.

```
for (var i in location) {
  if (typeof location[i] === "string") {
    console.log(i + ' = "' + location[i] + '"');
  }
}
   href = "http://search.phpied.com:8080/search?q=java&what=script#results"
   hash = "#results"
   host = "search.phpied.com:8080"
   hostname = "search.phpied.com"
   pathname = "/search"
   port = «8080»
   protocol = «http:»
   search = "?q=java&what=script"
```

There are also three methods that `location` provides, namely `reload()`, `assign()`, and `replace()`.

It's curious how many different ways exist for you to navigate to another page. Following are a few ways:

```
> window.location.href = 'http://www.packtpub.com';
> location.href = 'http://www.packtpub.com';
> location = 'http://www.packtpub.com';
> location.assign('http://www.packtpub.com');
```

`replace()` is almost the same as `assign()`. The difference is that it doesn't create an entry in the browser's history list as follows:

```
> location.replace('http://www.yahoo.com');
```

To reload a page you can use the following code:

```
> location.reload();
```

Alternatively, you can use `location.href` to point it to itself as follows:

```
> window.location.href = window.location.href;
```

Or, simply use the following code:

```
> location = location;
```

window.history

`window.history` allows limited access to the previously visited pages in the same browser session. For example, you can see how many pages the user has visited before coming to your page as follows:

```
> window.history.length;
  5
```

You cannot see the actual URLs though. For privacy reasons this doesn't work. See the following code:

```
> window.history[0];
```

You can, however, navigate back and forth through the user's session as if the user had clicked on the Back/Forward browser buttons as follows:

```
> history.forward();
> history.back();
```

You can also skip pages back and forth with `history.go()`. This is the same as calling `history.back()`. Code for history go() is as follows:

```
> history.go(-1);
```

For going two pages back use the following code:

```
> history.go(-2);
```

Reload the current page using the following code:

```
> history.go(0);
```

More recent browsers also support HTML5 History API, which lets you change the URL without reloading the page. This is perfect for dynamic pages because they can allow users to bookmark a specific URL, which represents the state of the application, and when they come back (or share with their friends) the page can restore the application state based on the URL. To get a sense of the history API, go to any page and write the following code in the console:

```
> history.pushState({a: 1}, "", "hello");

> history.pushState({b: 2}, "", "hello-you-too");

> history.state;
```

Notice how the URL changes, but the page is the same. Now, experiment with Back and Forward buttons in the browser and inspect the `history.state` again.

window.frames

`window.frames` is a collection of all of the frames in the current page. It doesn't distinguish between frames and iframes (inline frames). Regardless of whether there are frames on the page or not, `window.frames` always exists and points to `window` as follows:

```
> window.frames === window;
  true
```

Let's consider an example where you have a page with one iframe as follows:

```
<iframe name="myframe" src="hello.html" />
```

In order to tell if there are any frames on the page, you can check the `length` property. In case of one iframe, you'll see the following output:

```
> frames.length
  1
```

Each frame contains another page, which has its own global `window` object.

To get access to the iframe's `window`, you can do any of the following:

```
> window.frames[0];
> window.frames[0].window;
> window.frames[0].window.frames;
> frames[0].window;
> frames[0];
```

From the parent page, you can access properties of the child frame also. For example, you can reload the frame as follows:

```
> frames[0].window.location.reload();
```

From inside the child you can access the parent as follows:

```
> frames[0].parent === window;
  true
```

Using a property called `top`, you can access the top-most page (the one that contains all the other frames) from within any frame as follows:

```
> window.frames[0].window.top === window;
  true

> window.frames[0].window.top === window.top;
  true

> window.frames[0].window.top === top;
  true
```

In addition, `self` is the same as `window` as follows:

```
> self === window;
  true

> frames[0].self == frames[0].window;
  true
```

If a frame has a `name` attribute, you can not only access the frame by name, but also by index as follows:

```
> window.frames['myframe'] === window.frames[0];
  true
```

Or, alternatively you can use the following code:

```
> frames.myframe === window.frames[0];
  true
```

window.screen

`screen` provides information about the environment outside the browser. For example, the property `screen.colorDepth` contains the color bit-depth (the color quality) of the monitor. This is mostly used for statistical purposes. Have a look at the following code:

```
> window.screen.colorDepth;
  32
```

You can also check the available screen real estate (the resolution):

```
> screen.width;
  1440

> screen.availWidth;
  1440

> screen.height;
  900

> screen.availHeight;
  847
```

The difference between `height` and `availHeight` is that the `height` is the whole screen, while `availHeight` subtracts any operating system menus such as the Windows task bar. The same is the case for `width` and `availWidth`.

Somewhat related is the property mentioned in the following code:

```
> window.devicePixelRatio;
  1
```

It tells you the difference (ratio) between physical pixels and device pixels in the retina displays in mobile devices (for example, value 2 in iPhone).

window.open()/close()

Having explored some of the most common cross-browser properties of the `window` object, let's move to some of the methods. One such method is `open()`, which allows you to open new browser windows (pop ups). Various browser policies and user settings may prevent you from opening a pop up (due to abuse of the technique for marketing purposes), but generally you should be able to open a new window if it was initiated by the user. Otherwise, if you try to open a pop up as the page loads, it will most likely be blocked, because the user didn't initiate it explicitly.

`window.open()` accepts the following parameters:

- URL to load in the new window
- Name of the new window, which can be used as the value of a form's `target` attribute
- Comma-separated list of features. They are as follows:
    - `resizable`: Should the user be able to resize the new window
    - `width`, `height`: Width and height of the pop up
    - `status`: Should the status bar be visible

`window.open()` returns a reference to the `window` object of the newly created browser instance. Following is an example:

```
    var win = window.open('http://www.packtpub.com', 'packt',
                          'width=300,height=300,resizable=yes');
```

`win` points to the `window` object of the pop up. You can check if `win` has a falsy value, which means that the pop up was blocked.

`win.close()` closes the new window.

It's best to stay away from opening new windows for accessibility and usability reasons. If you don't like sites popping up windows to you, why do it to your users? There are legitimate purposes, such as providing help information while filling out a form, but often the same can be achieved with alternative solutions, such as using a floating `<div>` inside the page.

window.moveTo() and window.resizeTo()

Continuing with the shady practices from the past, following are more methods to irritate your users, provided their browser and personal settings allow you to.

- `window.moveTo(100, 100)` moves the browser window to screen location `x = 100` and `y = 100` (counted from the top-left corner)

- `window.moveBy(10, -10)` moves the window 10 pixels to the right and 10 pixels up from its current location
- `window.resizeTo(x, y)` and `window.resizeBy(x, y)` accept the same parameters as the move methods but they resize the window as opposed to moving it

Again, try to solve the problem you're facing without resorting to these methods.

window.alert(), window.prompt(), and window.confirm()

Chapter 2, *Primitive Data Types, Arrays, Loops, and Conditions*, talked about the function `alert()`. Now you know that global functions are accessible as methods of the global object so `alert('Watch out!')` and `window.alert('Watch out!')` are exactly the same.

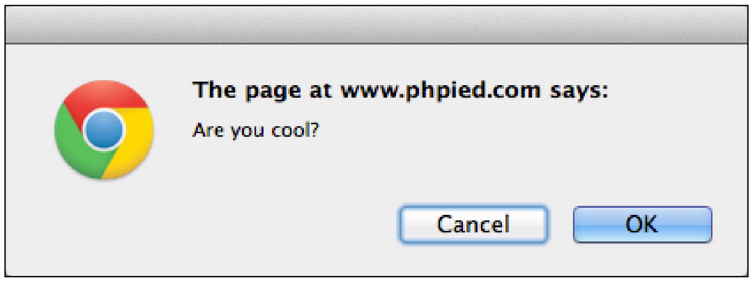
`alert()` is not an ECMAScript function, but a BOM method. In addition to it, two other BOM methods allow you to interact with the user through system messages. Following are the methods:

- `confirm()` gives the user two options, **OK** and **Cancel**
- `prompt()` collects textual input

See how this works as follows:

```
> var answer = confirm('Are you cool?');
> answer;
```

It presents you with a window similar to the following screenshot (the exact look depends on the browser and the operating system):


The page at www.phpied.com says:

Are you cool?

You'll notice the following things:

- Nothing gets written to the console until you close this message, this means that any JavaScript code execution freezes, waiting for the user's answer
- Clicking on **OK** returns **true**, clicking on **Cancel** or closing the message using the **X** icon (or the *ESC* key) returns **false**

This is handy for confirming user actions as follows:

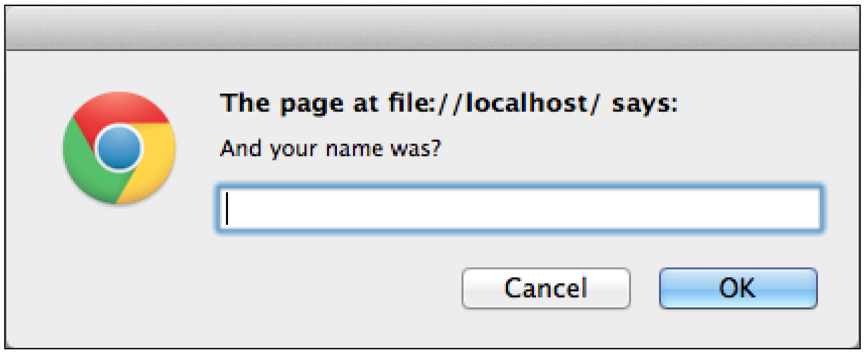
```
if (confirm('Sure you want to delete this?')) {
  // delete
} else {
  // abort
}
```

Make sure you provide an alternative way to confirm user actions for people who have disabled JavaScript (or for search engine spiders).

`window.prompt()` presents the user with a dialog to enter text as follows:

```
> var answer = prompt('And your name was?');
> answer;
```

This results in the following dialog box (Chrome, MacOS):


The page at file://localhost/ says:

And your name was?

The value of `answer` is one of the following:

- **null** if you click on **Cancel** or the **X** icon, or press *ESC*
- **""** (empty string) if you click on **OK** or press *Enter* without typing anything
- A text string if you type something and then click on **OK** (or press *Enter*)

The function also takes a string as a second parameter and displays it as a default value prefilled into the input field.

window.setTimeout() and window.setInterval()

`setTimeout()` and `setInterval()` allow for scheduling the execution of a piece of code. `setTimeout()` attempts to execute the given code once after a specified number of milliseconds. `setInterval()` attempts to execute it repeatedly after a specified number of milliseconds has passed.

This shows an alert after approximately 2 seconds (2000 milliseconds):

```
> function boo() { alert('Boo!'); }
> setTimeout(boo, 2000);
  4
```

As you can see the function returned an integer (in this case **4**) representing the ID of the timeout. You can use this ID to cancel the timeout using `clearTimeout()`. In the following example, if you're quick enough, and clear the timeout before 2 seconds have passed, the alert will never be shown as you can see in the following code:

```
> var id = setTimeout(boo, 2000);
> clearTimeout(id);
```

Let's change `boo()` to something less intrusive as follows:

```
> function boo() { console.log('boo'); }
```

Now, using `setInterval()` you can schedule `boo()` to execute every 2 seconds, until you cancel the scheduled execution with `clearInterval()`:

```
> var id = setInterval(boo, 2000);
  boo
  boo
  boo
  boo
  boo
  boo
> clearInterval(id);
```

Note, that both functions accept a pointer to a callback function as a first parameter. They can also accept a string which is evaluated with `eval()` but as you know, `eval()` is evil, so it should be avoided. And what if you want to pass arguments to the function? In such cases, you can just wrap the function call inside another function.

The following code is valid, but not recommended:

```
// bad idea
var id = setInterval("alert('boo, boo')", 2000);
```

This alternative is preferred:

```
var id = setInterval(
  function () {
    alert('boo, boo');
  },
  2000
);
```

Be aware that scheduling a function in some amount of milliseconds is not a guarantee that it will execute exactly at that time. One reason is that most browsers don't have millisecond resolution time. If you schedule something in 3 milliseconds, it will execute after a minimum of 15 in older IEs and sooner in more modern browsers, but most likely not in 1 millisecond. The other reason is that browsers maintain a queue of what you request them to do. 100 milliseconds timeout means add to the queue after 100 milliseconds. But if the queue is delayed by something slow happening, your function will have to wait and execute after, say, 120 milliseconds.

More recent browsers implement the `requestAnimationFrame()` function. It's preferable to the timeout functions because you're asking the browser to call your function whenever it has available resources, not after a predefined time in milliseconds. Try the following in your console:

```
function animateMe() {
  webkitRequestAnimationFrame(function(){
    console.log(new Date());
    animateMe();
  });
}

animateMe();
```

window.document

`window.document` is a BOM object that refers to the currently loaded document (page). Its methods and properties fall into the DOM category of objects. Take a deep breath (and maybe first look at the BOM exercises at the end of the chapter) and let's dive into the DOM.

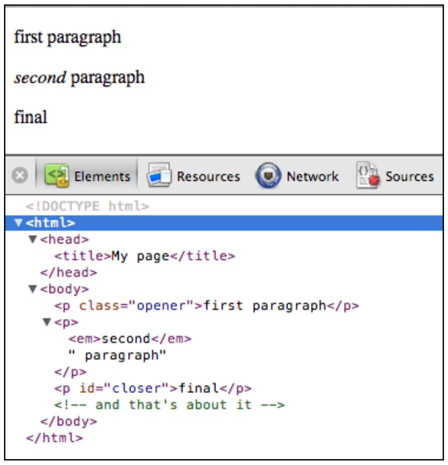
DOM

The Document Object Model (DOM) represents an XML or an HTML document as a tree of nodes. Using DOM methods and properties, you can access any element on the page, modify or remove elements, or add new ones. The DOM is a language-independent API (Application Programming Interface) and can be implemented not only in JavaScript, but also in any other language. For example, you can generate pages on the server-side with PHP's DOM implementation (`http://php.net/dom`).

Take a look at this example HTML page:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My page</title>
  </head>
  <body>
    <p class="opener">first paragraph</p>
    <p><em>second</em> paragraph</p>
    <p id="closer">final</p>
    <!-- and that's about it -->
  </body>
</html>
```

Consider the second paragraph (`<p><em>second</em> paragraph</p>`). You see that it's a `<p>` tag and it's contained in the `<body>` tag. If you think in terms of family relationships, you can say that `<body>` is the parent of `<p>` and `<p>` is the child. The first and the third paragraphs would also be children of the `<body>`, and at the same time siblings of the second paragraph. The `<em>` tag is a child of the second `<p>`, so `<p>` is its parent. The parent-child relationships can be represented graphically in an ancestry tree, called the DOM tree:


first paragraph

second paragraph

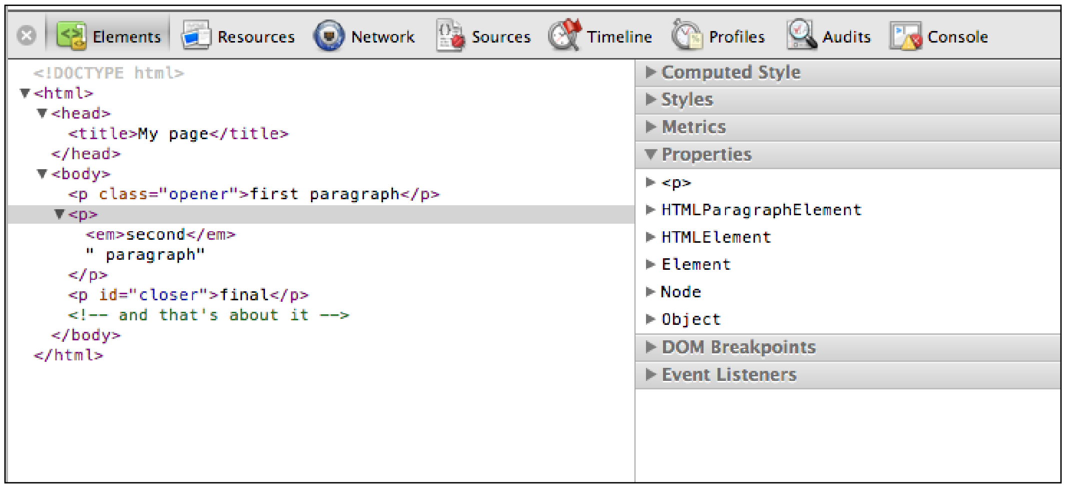
final

```
<!DOCTYPE html>
▼<html>
  ▼<head>
     <title>My page</title>
   </head>
  ▼<body>
     <p class="opener">first paragraph</p>
    ▼<p>
       <em>second</em>
       " paragraph"
     </p>
     <p id="closer">final</p>
     <!-- and that's about it -->
   </body>
 </html>
```

The previous screenshot shows what you'll see in the webkit console's **Elements** tab after you expand each node.

You can see how all of the tags are shown as expandable nodes on the tree. Although not shown, there exists the so-called text nodes, for example, the text inside the `<em>` (the word second) is a text node. Whitespace is also considered a text node. Comments inside the HTML code are also nodes in the tree, the `<!-- and that's about it -->` comment in the HTML source is a comment node in the tree.

Every node in the DOM tree is an object and the **Properties** section on the right lists all of the properties and methods you can use to work with these objects, following the inheritance chain of how this object was created:



```
<!DOCTYPE html>
▼<html>
  ▼<head>
     <title>My page</title>
   </head>
  ▼<body>
     <p class="opener">first paragraph</p>
    ▼<p>
       <em>second</em>
       " paragraph"
     </p>
     <p id="closer">final</p>
     <!-- and that's about it -->
   </body>
</html>
```

```
▶ Computed Style
▶ Styles
▶ Metrics
▼ Properties
  ▶ <p>
  ▶ HTMLParagraphElement
  ▶ HTMLElement
  ▶ Element
  ▶ Node
  ▶ Object
▶ DOM Breakpoints
▶ Event Listeners
```

You can also see the constructor function that was used behind the scenes to create each of these objects. Although this is not too practical for day-to-day tasks, it may be interesting to know that, for example, `<p>` is created by the `HTMLParagraphElement()` constructor, the object that represents the `head` tag is created by `HTMLHeadElement()`, and so on. You cannot create objects using these constructors directly, though.

Core DOM and HTML DOM

One last diversion before moving on to more practical examples. As you now know, the DOM represents both XML documents and HTML documents. In fact, HTML documents are XML documents, but a little more specific. Therefore, as part of DOM Level 1, there is a Core DOM specification that is applicable to all XML documents, and there is also an HTML DOM specification, which extends and builds upon the core DOM. Of course, the HTML DOM doesn't apply to all XML documents, but only to HTML documents. Let's see some examples of Core DOM and HTML DOM constructors:

Constructor	**Inherits from**	**Core or HTML**	**Comment**
`Node`		Core	Any node on the tree.
`Document`	`Node`	Core	The `document` object, the main entry point to any XML document.
`HTMLDocument`	`Document`	HTML	This is `window.document` or simply `document`, the HTML-specific version of the previous object, which you'll use extensively.
`Element`	`Node`	Core	Every tag in the source is represented by an element. That's why you say "the P element" meaning "the `<p></p>` tag".
`HTMLElement`	`Element`	HTML	General-purpose constructor, all constructors for HTML elements inherit from it.
`HTMLBodyElement`	`HTMLElement`	HTML	Element representing the `<body>` tag.
`HTMLLinkElement`	`HTMLElement`	HTML	An A element (an `<a href="..."></a>` tag).
and other such constructors.	`HTMLElement`	HTML	All the rest of the HTML elements.
`CharacterData`	`Node`	Core	General-purpose constructor for dealing with texts.
`Text`	`CharacterData`	Core	Text node inside a tag. In `<em>second</em>` you have the element node EM and the text node with value second.
`Comment`	`CharacterData`	Core	`<!-- any comment -->`

Constructor	Inherits from	Core or HTML	Comment
`Attr`	`Node`	Core	Represents an attribute of a tag, in `<p id="closer">` the `id` attribute is a DOM object created by the `Attr()` constructor.
`NodeList`		Core	A list of nodes, an array-like object that has a `length` property.
`NamedNodeMap`		Core	Same as `NodeList` but the nodes can be accessed by name, not only by numeric index.
`HTMLCollection`		HTML	Similar to `NamedNodeMap` but specific for HTML.

These are by no means all of the Core DOM and HTML DOM objects. For the full list consult `http://www.w3.org/TR/DOM-Level-1/`.

Now that this bit of DOM theory is behind you, let's focus on the practical side of working with the DOM. In the following sections, you'll learn how to do the following things:

- Access DOM nodes
- Modify nodes
- Create new nodes
- Remove nodes

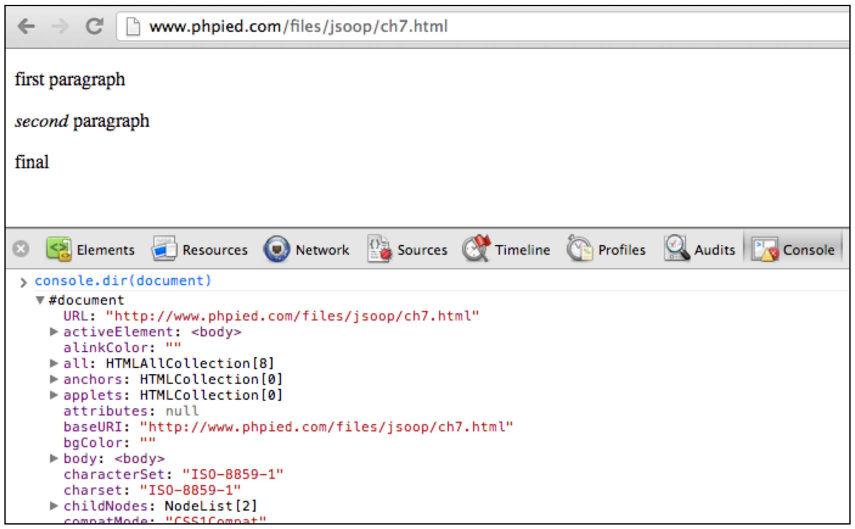
Accessing DOM nodes

Before you can validate the user input in a form on a page or swap an image, you need to get access to the element you want to inspect or modify. Luckily, there are many ways to get to any element, either by navigating around traversing the DOM tree or by using a shortcut.

It's best if you start experimenting with all of the new objects and methods. The examples you'll see use the same simple document that you saw at the beginning of the DOM section, and which you can access at `http://www.phpied.com/files/jsoop/ch7.html`. Open your console, and let's get started.

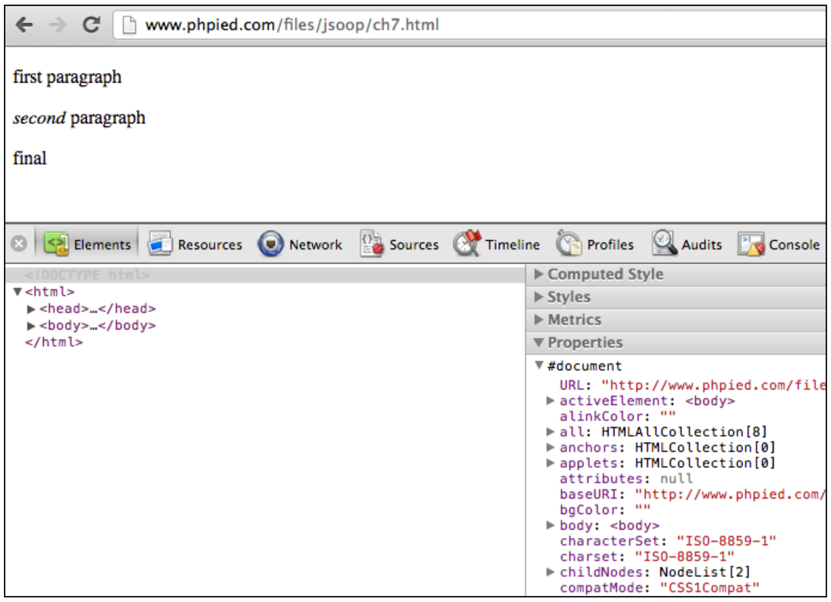
The document node

`document` gives you access to the current document. To explore this object, you can use your console as a cheat sheet. Type `console.dir(document)` and click on the result:



```
> console.dir(document)
  ▼#document
     URL: "http://www.phpied.com/files/jsoop/ch7.html"
    ▶activeElement: <body>
     alinkColor: ""
    ▶all: HTMLAllCollection[8]
    ▶anchors: HTMLCollection[0]
    ▶applets: HTMLCollection[0]
     attributes: null
     baseURI: "http://www.phpied.com/files/jsoop/ch7.html"
     bgColor: ""
    ▶body: <body>
     characterSet: "ISO-8859-1"
     charset: "ISO-8859-1"
    ▶childNodes: NodeList[2]
```

Alternatively, you can browse all of the properties and methods of the `document` object DOM properties in the **Elements** panel:



```
<!DOCTYPE html>
▼<html>
  ▶<head>…</head>
  ▶<body>…</body>
 </html>
```

```
▼#document
   URL: "http://www.phpied.com/file
  ▶activeElement: <body>
   alinkColor: ""
  ▶all: HTMLAllCollection[8]
  ▶anchors: HTMLCollection[0]
  ▶applets: HTMLCollection[0]
   attributes: null
   baseURI: "http://www.phpied.com/
   bgColor: ""
  ▶body: <body>
   characterSet: "ISO-8859-1"
   charset: "ISO-8859-1"
  ▶childNodes: NodeList[2]
   compatMode: "CSS1Compat"
```

All nodes (this also includes the document node, text nodes, element nodes, and attribute nodes) have `nodeType`, `nodeName`, and `nodeValue` properties:

```
> document.nodeType;
  9
```

There are 12 node types, represented by integers. As you can see, the document node type is **9**. The most commonly used are 1 (element), 2 (attribute), and 3 (text).

Nodes also have names. For HTML tags the node name is the tag name (`tagName` property). For text nodes, it's **#text**, and for document nodes the name is as follows:

```
> document.nodeName;
  "#document"
```

Nodes can also have node values. For example, for text nodes the value is the actual text. The document node doesn't have a value which can be seen as follows:

```
> document.nodeValue;
  null
```

documentElement

Now, let's move around the tree. XML documents always have one root node that wraps the rest of the document. For HTML documents, the root is the `<html>` tag. To access the root, you use the `documentElement` property of the `document` object:

```
> document.documentElement;
  <html>...</html>
```

`nodeType` is **1** (an element node) which can be seen as follows:

```
> document.documentElement.nodeType;
  1
```

For element nodes, both `nodeName` and `tagName` properties contain the name of the tag, as seen in the following output:

```
> document.documentElement.nodeName;
  "HTML"

> document.documentElement.tagName;
  "HTML"
```

Child nodes

In order to tell if a node has any children you use `hasChildNodes()` as follows:

```
> document.documentElement.hasChildNodes();
  true
```

The HTML element has three children, the `head` and the `body` elements and the whitespace between them (whitespace is counted in most, but not all browsers). You can access them using the `childNodes` array-like collection as follows:

```
> document.documentElement.childNodes.length;
  3

> document.documentElement.childNodes[0];
  <head>...</head>

> document.documentElement.childNodes[1];
  #text

> document.documentElement.childNodes[2];
  <body>...</body>
```

Any child has access to its parent through the `parentNode` property, as seen in the following code:

```
> document.documentElement.childNodes[1].parentNode;
  <html>...</html>
```

Let's assign a reference to `body` to a variable as follows:

```
> var bd = document.documentElement.childNodes[2];
```

How many children does the `body` element have?

```
> bd.childNodes.length;
  9
```

As a refresher, here again is the `body` of the document:

```
<body>
  <p class="opener">first paragraph</p>
  <p><em>second</em> paragraph</p>
  <p id="closer">final</p>
  <!-- and that's about it -->
</body>
```

How come `body` has **9** children? Well, three paragraphs plus one comment makes four nodes. The whitespace between these four nodes makes three more text nodes. This makes a total of seven so far. The whitespace between `<body>` and the first `<p>` is the eighth node. The whitespace between the comment and the closing `</body>` is another text node. This makes a total of nine child nodes. Just type `bd.childNodes` in the console to inspect them all.

Attributes

Because the first child of the body is a whitespace, the second child (index 1) is the first paragraph. Refer to the following piece of code:

```
> bd.childNodes[1];
    <p class="opener">first paragraph</p>
```

You can check whether an element has attributes using `hasAttributes()` as follows:

```
> bd.childNodes[1].hasAttributes();
    true
```

How many attributes? In this example, one is the `class` attribute which can be seen as follows:

```
> bd.childNodes[1].attributes.length;
    1
```

You can access the attributes by index and by name. You can also get the value using the `getAttribute()` method as follows:

```
> bd.childNodes[1].attributes[0].nodeName;
    "class"

> bd.childNodes[1].attributes[0].nodeValue;
    "opener"

> bd.childNodes[1].attributes['class'].nodeValue;
    "opener"

> bd.childNodes[1].getAttribute('class');
    "opener"
```

Accessing the content inside a tag

Let's take a look at the first paragraph:

```
> bd.childNodes[1].nodeName;
    "P"
```

You can get the text contained in the paragraph by using the `textContent` property. `textContent` doesn't exist in older IEs, but another property called `innerText` returns the same value, as seen in the following output:

```
> bd.childNodes[1].textContent;
  "first paragraph"
```

There is also the `innerHTML` property. It's a relatively new addition to the DOM standard despite the fact that it previously existed in all major browsers. It returns (or sets) HTML code contained in a node. You can see how this is a little inconsistent as DOM treats the document as a tree of nodes, not as a string of tags. But `innerHTML` is so convenient to use that you'll see it everywhere. Refer to the following code:

```
> bd.childNodes[1].innerHTML;
  "first paragraph"
```

The first paragraph contains only text, so `innerHTML` is the same as `textContent` (or `innerText` in IE). However, the second paragraph does contain an `em` node, so you can see the difference as follows:

```
> bd.childNodes[3].innerHTML;
  "<em>second</em> paragraph"

> bd.childNodes[3].textContent;
  "second paragraph"
```

Another way to get the text contained in the first paragraph is by using the `nodeValue` of the text node contained inside the `p` node as follows:

```
> bd.childNodes[1].childNodes.length;
   1

> bd.childNodes[1].childNodes[0].nodeName;
  "#text"

> bd.childNodes[1].childNodes[0].nodeValue;
  "first paragraph"
```

DOM access shortcuts

By using `childNodes`, `parentNode`, `nodeName`, `nodeValue`, and `attributes` you can navigate up and down the tree and do anything with the document. But the fact that whitespace is a text node makes this a fragile way of working with the DOM. If the page changes, your script may no longer work correctly. Also, if you want to get to a node deeper in the tree, it could take a bit of code before you get there. That's why you have shortcut methods, namely, `getElementsByTagName()`, `getElementsByName()`, and `getElementById()`.

`getElementsByTagName()` takes a tag name (the name of an element node) and returns an HTML collection (array-like object) of nodes with the matching tag name. For example, the following example asks "give me a count of all paragraphs" which is given as follows:

```
> document.getElementsByTagName('p').length;
   3
```

You can access an item in the list by using the brackets notation, or the method `item()`, and passing the index (0 for the first element). Using `item()` is discouraged as array brackets are more consistent and also shorter to type. Refer to the following piece of code:

```
> document.getElementsByTagName('p')[0];
   <p class="opener">first paragraph</p>

> document.getElementsByTagName('p').item(0);
   <p class="opener">first paragraph</p>
```

Getting the contents of the first `p` can be done as follows:

```
> document.getElementsByTagName('p')[0].innerHTML;
  "first paragraph"
```

Accessing the last `p` can be done as follows:

```
> document.getElementsByTagName('p')[2];
   <p id="closer">final</p>
```

To access the attributes of an element, you can use the `attributes` collection or `getAttribute()` as shown previously. But a shorter way is to use the attribute name as a property of the element you're working with. So to get the value of the `id` attribute, you just use `id` as a property as follows:

```
> document.getElementsByTagName('p')[2].id;
  "closer"
```

Getting the `class` attribute of the first paragraph won't work though. It's an exception, because it just happens so that class is a reserved word in ECMAScript. You can use `className` instead as follows:

```
> document.getElementsByTagName('p')[0].className;
  "opener"
```

Using `getElementsByTagName()` you can get all of the elements on the page as follows:

```
> document.getElementsByTagName('*').length;
    8
```

In earlier versions of IE before IE7, `*` is not acceptable as a tag name. To get all elements you can use IE's proprietary `document.all` collection, although selecting every element is rarely needed.

The other shortcut mentioned is `getElementById()`. This is probably the most common way of accessing an element. You just assign IDs to the elements you plan to play with and they'll be easy to access later on, as seen in the following code:

```
> document.getElementById('closer');
<p id="closer">final</p>
```

Additional shortcut methods in more recent browsers include the following:

- `getElementByClassName()`: This method finds elements using their `class` attribute
- `querySelector()`: This method finds an element using a CSS selector string
- `querySelectorAll()`: This method is the same as the previous one but returns all matching elements not just the first

Siblings, body, first, and last child

`nextSibling` and `previousSibling` are two other convenient properties to navigate the DOM tree, once you have a reference to one element:

```
> var para = document.getElementById('closer');
> para.nextSibling;
    #text

> para.previousSibling;
    #text

> para.previousSibling.previousSibling;
     <p>...</p>

> para.previousSibling.previousSibling.previousSibling;
    #text

> para.previousSibling.previousSibling.nextSibling.nextSibling;
     <p id="closer">final</p>
```

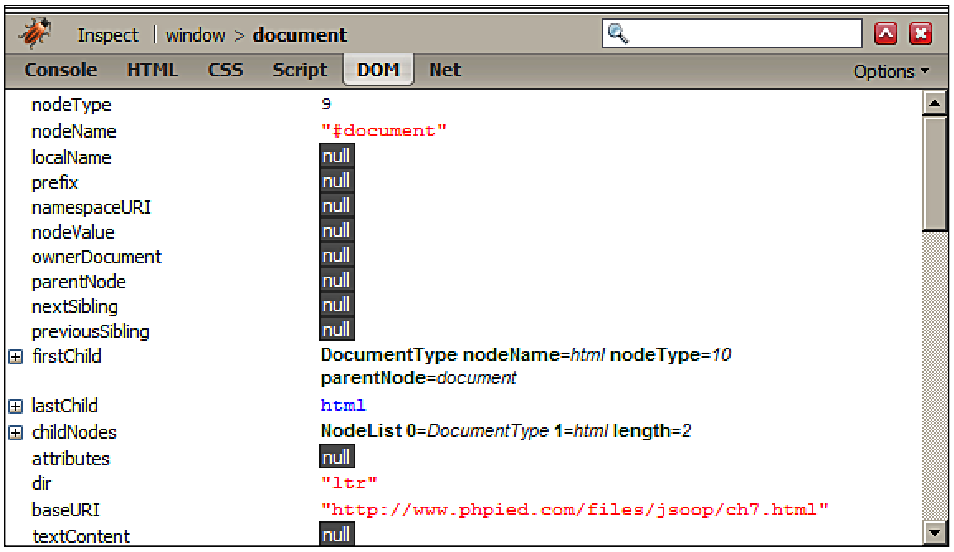
The body element is used so often that it has its own shortcut:

```
> document.body;
   <body>...</body>

> document.body.nextSibling;
   null

> document.body.previousSibling.previousSibling;
   <head>...</head>
```

`firstChild` and `lastChild` are also convenient. `firstChild` is the same as `childNodes[0]` and `lastChild` is the same as `childNodes[childNodes.length - 1]`:

```
> document.body.firstChild;
  #text

> document.body.lastChild;
  #text

> document.body.lastChild.previousSibling;
   <!-- and that's about it -->

> document.body.lastChild.previousSibling.nodeValue;
  " and that's about it "
```

The following screenshot shows the family relationships between the body and three paragraphs in it. For simplicity, all the whitespace text nodes are removed from the screenshot:



```
nodeType          9
nodeName          "#document"
localName         null
prefix            null
namespaceURI      null
nodeValue         null
ownerDocument     null
parentNode        null
nextSibling       null
previousSibling   null
⊞ firstChild      DocumentType nodeName=html nodeType=10
                  parentNode=document
⊞ lastChild       html
⊞ childNodes      NodeList 0=DocumentType 1=html length=2
attributes        null
dir               "ltr"
baseURI           "http://www.phpied.com/files/jsoop/ch7.html"
textContent       null
```

Walk the DOM

To wrap up, here's a function that takes any node and walks through the DOM tree recursively, starting from the given node:

```
function walkDOM(n) {
  do {
    console.log(n);
    if (n.hasChildNodes()) {
      walkDOM(n.firstChild);
    }
  } while (n = n.nextSibling);
}
```

You can test the function as follows:

```
> walkDOM(document.documentElement);
> walkDOM(document.body);
```

Modifying DOM nodes

Now that you know a whole lot of methods for accessing any node of the DOM tree and its properties, let's see how you can modify these nodes.

Let's assign a pointer to the last paragraph to the variable `my` as follows:

```
> var my = document.getElementById('closer');
```

Now, changing the text of the paragraph can be as easy as changing the `innerHTML` value:

```
> my.innerHTML = 'final!!!';
   "final!!!"
```

Because `innerHTML` accepts a string of HTML source code, you can also create a new `em` node in the DOM tree as follows:

```
> my.innerHTML = '<em>my</em> final';
   "<em>my</em> final"
```

The new `em` node becomes a part of the tree:

```
> my.firstChild;
   <em>my</em>

> my.firstChild.firstChild;
   "my"
```

Another way to change text is to get the actual text node and change its `nodeValue` as follows:

```
> my.firstChild.firstChild.nodeValue = 'your';
  "your"
```

Modifying styles

Often you don't change the content of a node but its presentation. The elements have a `style` property, which in turn has a property mapped to each CSS property. For example, changing the style of the paragraph to add a red border:

```
> my.style.border = "1px solid red";
  "1px solid red"
```

CSS properties often have dashes, but dashes are not acceptable in JavaScript identifiers. In such cases, you skip the dash and uppercase the next letter. So `padding-top` becomes `paddingTop`, `margin-left` becomes `marginLeft`, and so on. Have a look at the following code:

```
> my.style.fontWeight = 'bold';
  "bold"
```

You also have access to `cssText` property of `style`, which lets you work with styles as strings:

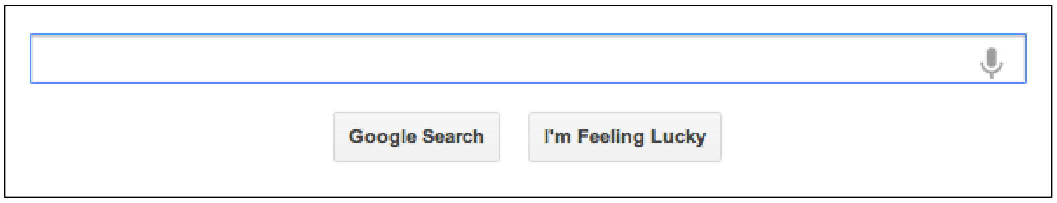
```
> my.style.cssText;
  "border: 1px solid red; font-weight: bold;"
```

And modifying styles is a string manipulation:

```
> my.style.cssText += " border-style: dashed;"
"border: 1px dashed red; font-weight: bold; border-style: dashed;"
```

Fun with forms

As mentioned earlier, JavaScript is great for client-side input validation and can save a few round-trips to the server. Let's practice form manipulations and play a little bit with a form located on a popular page `www.google.com`:



Finding the first text input using the `querySelector()` method and a CSS selector string is as follows:

```
> var input = document.querySelector('input[type=text]');
```

Accessing the search box:

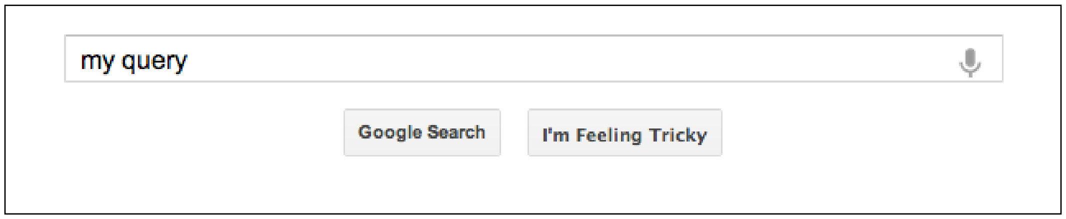
```
> input.name;
```
    "q"

Changing the search query by setting the text contained in the `value` attribute is done as follows:

```
> input.value = 'my query';
```
    "my query"

Now, let's have some fun. Changing the word **Lucky** with **Tricky** in the button:

```
> var feeling = document.querySelectorAll("button")[2];
> feeling.textContent = feelingtextContent.replace(/Lu/, 'Tri');
```
    "I'm Feeling Tricky"



Now, let's implement the tricky part and make that button show and hide for one second. You can do this with a simple function. Let's call it `toggle()`. Every time you call the function, it checks the value of the CSS property `visibility` and sets it to visible if it's hidden and vice versa using following code:

```
function toggle() {
  var st = document.querySelectorAll('button')[2].style;
  st.visibility = (st.visibility === 'hidden')
    ? 'visible'
    : 'hidden';
}
```

Instead of calling the function manually, let's set an interval and call it every second:

```
> var myint = setInterval(toggle, 1000);
```

The result? The button starts blinking (making it trickier to click). When you're tired of chasing it, just remove the timeout interval:

```
> clearInterval(myint);
```

Creating new nodes

To create new nodes, you can use the methods `createElement()` and `createTextNode()`. Once you have the new nodes, you add them to the DOM tree using `appendChild()` (or `insertBefore()`, or `replaceChild()`).

Reload `http://www.phpied.com/files/jsoop/ch7.html` and let's play.

Creating a new `p` element and set its `innerHTML`, as shown in the following code:

```
> var myp = document.createElement('p');
> myp.innerHTML = 'yet another';
  "yet another"
```

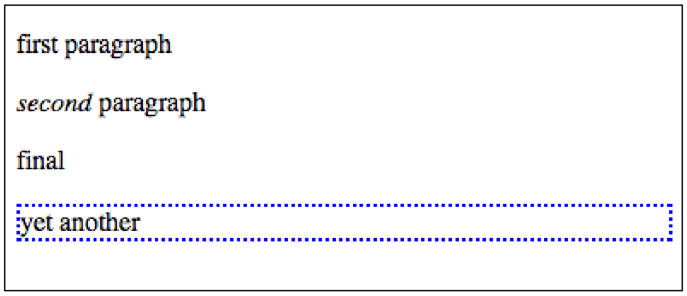
The new element automatically gets all the default properties, such as `style`, which you can modify:

```
> myp.style;
  CSSStyleDeclaration

> myp.style.border = '2px dotted blue';
  "2px dotted blue"
```

Using `appendChild()` you can add the new node to the DOM tree. Calling this method on the `document.body` node means creating one more child node right after the last child:

```
> document.body.appendChild(myp);
  <p style="border: 2px dotted blue;">yet another</p>
```

Here's an illustration of how the page looks like after the new node is appended:



first paragraph

second paragraph

final

yet another

DOM-only method

`innerHTML` gets things done a little more quickly than using pure DOM. In pure DOM you need to perform the following steps:

1. Create a new text node containing yet another text
2. Create a new paragraph node
3. Append the text node as a child to the paragraph
4. Append the paragraph as a child to the body

This way you can create any number of text nodes and elements and nest them however you like. Let's say you want to add the following HTML to the end of the body:

```
<p>one more paragraph<strong>bold</strong></p>
```

Presenting this as a hierarchy would be something like the following:

```
P element
    text node with value "one more paragraph"
    STRONG element
        text node with value "bold"
```

The code that accomplishes this is as follows:

```
// create P
var myp = document.createElement('p');
// create text node and append to P
var myt = document.createTextNode('one more paragraph');
myp.appendChild(myt);
// create STRONG and append another text node to it
var str = document.createElement('strong');
str.appendChild(document.createTextNode('bold'));
// append STRONG to P
myp.appendChild(str);
// append P to BODY
document.body.appendChild(myp);
```

cloneNode()

Another way to create nodes is by copying (or cloning) existing ones. The method `cloneNode()` does this and accepts a boolean parameter (`true` = deep copy with all the children, `false` = shallow copy, only this node). Let's test the method.

Getting a reference to the element you want to clone can be done as follows:

```
> var el = document.getElementsByTagName('p')[1];
```

Now, `el` refers to the second paragraph on the page that looks like the following code:

```
<p><em>second</em> paragraph</p>
```

Let's create a shallow clone of `el` and append it to the body:

```
> document.body.appendChild(el.cloneNode(false));
```

You won't see a difference on the page, because the shallow copy only copied the P node, without any children. This means that the text inside the paragraph (which is a text node child) was not cloned. The line above would be equivalent to the following:

```
> document.body.appendChild(document.createElement('p'));
```

But if you create a deep copy, the whole DOM subtree starting from P is copied, and this includes text nodes and the EM element. This line copies (visually too) the second paragraph to the end of the document:

```
> document.body.appendChild(el.cloneNode(true));
```

You can also copy only the EM if you want as follows:

```
> document.body.appendChild(el.firstChild.cloneNode(true));
  <em>second</em>
```

Or, only the text node with value second:

```
> document.body.appendChild(
    el.firstChild.firstChild.cloneNode(false));
  "second"
```

insertBefore()

Using `appendChild()`, you can only add new children at the end of the selected element. For more control over the exact location there is `insertBefore()`. This is the same as `appendChild()`, but accepts an extra parameter specifying where (before which element) to insert the new node. For example, the following code inserts a text node at the end of the `body`:

```
> document.body.appendChild(document.createTextNode('boo!'));
```

And this creates another text node and adds it as the first child of the `body`:

```
document.body.insertBefore(
  document.createTextNode('first boo!'),
  document.body.firstChild
);
```

Removing nodes

To remove nodes from the DOM tree, you can use the method `removeChild()`. Again, let's start fresh with the same page with the body:

```
<body>
  <p class="opener">first paragraph</p>
  <p><em>second</em> paragraph</p>
  <p id="closer">final</p>
  <!-- and that's about it -->
</body>
```

Here's how you can remove the second paragraph:

```
> var myp = document.getElementsByTagName('p')[1];
> var removed = document.body.removeChild(myp);
```

The method returns the removed node if you want to use it later. You can still use all the DOM methods even though the element is no longer in the tree:

```
> removed;
  <p>...</p>

> removed.firstChild;
  <em>second</em>
```

There's also the `replaceChild()` method that removes a node and puts another one in its place.

After removing the node, the tree looks like the following:

```
<body>
  <p class="opener">first paragraph</p>
  <p id="closer">final</p>
  <!-- and that's about it -->
</body>
```

Now, the second paragraph is the one with the ID "closer":

```
> var p = document.getElementsByTagName('p')[1];
> p;
```
<p id="closer">final</p>

Let's replace this paragraph with the one in the `removed` variable:

```
> var replaced = document.body.replaceChild(removed, p);
```

Just like `removeChild()`, `replaceChild()` returns a reference to the node that is now out of the tree:

```
> replaced;
```
<p id="closer">final</p>

Now, the body looks like the following:

```
<body>
  <p class="opener">first paragraph</p>
  <p><em>second</em> paragraph</p>
  <!-- and that's about it -->
</body>
```

A quick way to wipe out all of the content of a subtree is to set the `innerHTML` to a blank string. This removes all of the children of the BODY:

```
> document.body.innerHTML = '';
```
""

Testing is done as follows:

```
> document.body.firstChild;
```
null

Removing with `innerHTML` is fast and easy. The DOM-only way would be to go over all of the child nodes and remove each one individually. Here's a little function that removes all nodes from a given start node:

```
function removeAll(n) {
  while (n.firstChild) {
    n.removeChild(n.firstChild);
  }
}
```

If you want to delete all BODY children and leave the page with an empty `<body></body>` use the following code:

```
> removeAll(document.body);
```

HTML-only DOM objects

As you know already, the Document Object Model applies to both XML and HTML documents. What you've learned above about traversing the tree and then adding, removing, or modifying nodes applies to any XML document. There are, however, some HTML-only objects and properties.

`document.body` is one such HTML-only object. It's so common to have a `<body>` tag in HTML documents and it's accessed so often, that it makes sense to have an object that's shorter and friendlier than the equivalent `document.getElementsByTagName('body')[0]`.

`document.body` is one example of a legacy object inherited from the prehistoric DOM Level 0 and moved to the HTML extension of the DOM specification. There are other objects similar to `document.body`. For some of them there is no core DOM equivalent, for others there is an equivalent, but the DOM0 original was ported anyway for simplicity and legacy purposes. Let's see some of those objects.

Primitive ways to access the document

Unlike the DOM, which gives you access to any element (and even comments and whitespace), initially JavaScript had only limited access to the elements of an HTML document. This was done mainly through a number of collections:

- `document.images`: This is a collection of all of the images on the page. The Core DOM equivalent is `document.getElementsByTagName('img')`
- `document.applets`: This is the same as `document.getElementsByTagName('applet')`
- `document.links`
- `document.anchors`
- `document.forms`

`document.links` contains a list of all `<a href="..."></a>` tags on the page, meaning the `<a>` tags that have an `href` attribute. `document.anchors` contain all links with a `name` attribute (`<a name="..."></a>`).

One of the most widely used collections is `document.forms`, which contains a list of `<form>` elements.

Let's play with a page that contains a form and an input, `http://www.phpied.com/files/jsoop/ch7-form.html`. The following gives you access to the first form on the page:

```
> document.forms[0];
```

It's the same as the following:

```
> document.getElementsByTagName('forms')[0];
```

The `document.forms` collection contains collections of input fields and buttons, accessible through the `elements` property. Here's how to access the first input of the first form on the page:

```
> document.forms[0].elements[0];
```

Once you have access to an element, you can access its attributes as object properties. The first field of the first form in the test page is this:

```
<input name="search" id="search" type="text" size="50"
       maxlength="255" value="Enter email..." />
```

You can change the text in the field (the value of the `value` attribute) by using the following code:

```
> document.forms[0].elements[0].value = 'me@example.org';
   "me@example.org"
```

If you want to disable the field dynamically use the following code:

```
> document.forms[0].elements[0].disabled = true;
```

When forms or form elements have a `name` attribute, you can access them by name too as in the following code:

```
> document.forms[0].elements['search']; // array notation
> document.forms[0].elements.search;    // object property
```

document.write()

The method `document.write()` allows you to insert HTML into the page while the page is being loaded. You can have something like the following code:

```
<p>It is now
  <script>
    document.write("<em>" + new Date() + "</em>");
  </script>
</p>
```

This is the same as if you had the date directly in the source of the HTML document as follows:

```
<p>It is now
  <em>Fri Apr 26 2013 16:55:16 GMT-0700 (PDT)</em>
</p>
```

Note, that you can only use `document.write()` while the page is being loaded. If you try it after page load, it will replace the content of the whole page.

It's rare that you would need `document.write()`, and if you think you do, try an alternative approach. The ways to modify the contents of the page provided by DOM Level 1 are preferred and are much more flexible.

Cookies, title, referrer, domain

The four additional properties of `document` you'll see in this section are also ported from DOM Level 0 to the HTML extension of DOM Level 1. Unlike the previous ones, for these properties there are no core DOM equivalents.

`document.cookie` is a property that contains a string. This string is the content of the cookies exchanged between the server and the client. When the server sends a page to the browser, it may include the `Set-Cookie` HTTP header. When the client sends a request to the server, it sends the cookie information back with the `Cookie` header. Using `document.cookie` you can alter the cookies the browser sends to the server. For example, visiting `cnn.com` and typing `document.cookie` in the console gives you the following output:

```
> document.cookie;
  "mbox=check#true#1356053765|session#1356053704195-121286#1356055565;...
```

`document.title` allows you to change the title of the page displayed in the browser window. For example, see the following code:

```
> document.title = 'My title';
  "My title"
```

Note, that this doesn't change the value of the `<title>` element, but only the display in the browser window, so it's not equivalent to `document.querySelector('title')`.

`document.referrer` tells you the URL of the previously-visited page. This is the same value the browser sends in the `Referer` HTTP header when requesting the page. (Note, that `Referer` is misspelled in the HTTP headers, but is correct in JavaScript's `document.referrer`). If you've visited the CNN page by searching on Yahoo first, you can see something like the following:

```
> document.referrer;
  "http://search.yahoo.com/search?p=cnn&ei=UTF-8&fr=moz2"
```

`document.domain` gives you access to the domain name of the currently loaded page. This is commonly used when you need to perform so-called domain relaxation. Imagine your page is `www.yahoo.com` and inside it you have an iframe hosted on `music.yahoo.com` subdomain. These are two separate domains so the browser's security restrictions won't allow the page and the iframe to communicate. To resolve this you can set `document.domain` on both pages to `yahoo.com` and they'll be able to talk to each other.

Note, that you can only set the domain to a less-specific one, for example, you can change `www.yahoo.com` to `yahoo.com`, but you cannot change `yahoo.com` to `www.yahoo.com` or any other non-yahoo domain.

```
> document.domain;
    "www.yahoo.com"

> document.domain = 'yahoo.com';
    "yahoo.com"

> document.domain = 'www.yahoo.com';
    Error: SecurityError: DOM Exception 18

> document.domain = 'www.example.org';
    Error: SecurityError: DOM Exception 18
```

Previously in this chapter, you saw the `window.location` object. Well, the same functionality is also available as `document.location`:

```
> window.location === document.location;
    true
```

Events

Imagine you are listening to a radio program and they announce, "Big event! Huge! Aliens have landed on Earth!" You might think "Yeah, whatever", some other listeners might think "They come in peace" and some "We're all gonna die!". Similarly, the browser broadcasts events and your code could be notified should it decide to tune in and listen to the events as they happen. Some example events include:

- The user clicks a button
- The user types a character in a form field
- The page finishes loading

You can attach a JavaScript function (called an event listener or event handler) to a specific event and the browser will invoke your function as soon the event occurs. Let's see how this is done.

Inline HTML attributes

Adding specific attributes to a tag is the laziest (but the least maintainable) way, for example:

```
<div onclick="alert('Ouch!')">click</div>
```

In this case when the user clicks on the `<div>`, the click event fires and the string of JavaScript code contained in the `onclick` attribute is executed. There's no explicit function that listens to the click event, but behind the scenes a function is still created and it contains the code you specified as a value of the `onclick` attribute.

Element Properties

Another way to have some code executed when a click event fires is to assign a function to the `onclick` property of a DOM node element. For example:

```
<div id="my-div">click</div>
<script>
  var myelement = document.getElementById('my-div');
  myelement.onclick = function () {
    alert('Ouch!');
    alert('And double ouch!');
  };
</script>
```

This way is better because it helps you keep your `<div>` clean of any JavaScript code. Always keep in mind that HTML is for content, JavaScript for behavior, and CSS for formatting, and you should keep these three separate as much as possible.

This method has the drawback that you can attach only one function to the event, as if the radio program has only one listener. It's true that you can have a lot happening inside the same function, but this is not always convenient, as if all the radio listeners are in the same room.

DOM event listeners

The best way to work with browser events is to use the event listener approach outlined in DOM Level 2, where you can have many functions listening to an event. When the event fires, all functions are executed. All of the listeners don't need to know about each other and can work independently. They can tune in and out at any time without affecting the other listeners.

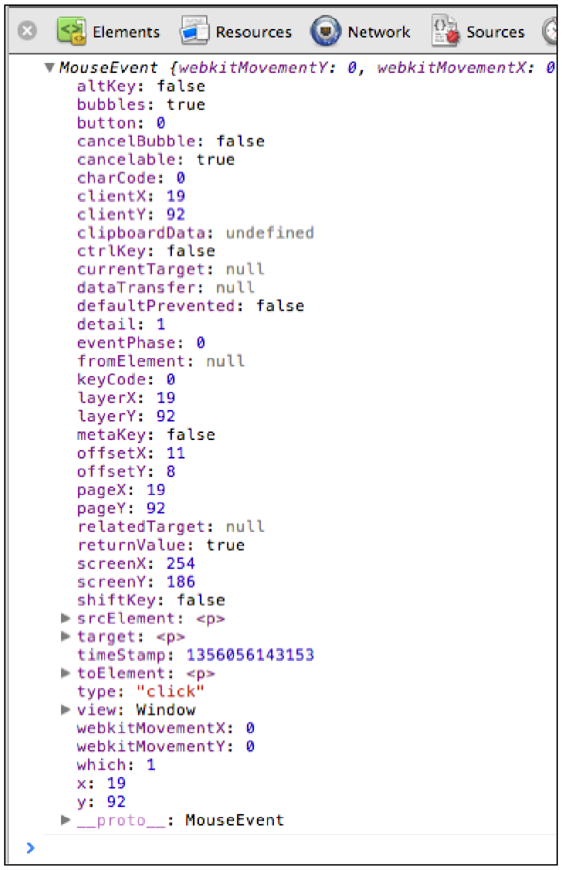
Let's use the same simple markup from the previous section (available for you to play with at `http://www.phpied.com/files/jsoop/ch7.html`). It has this piece of markup as follows:

```
<p id="closer">final</p>
```

Your JavaScript code can assign listeners to the click event using the `addEventListener()` method. Let's attach two listeners as follows:

```
var mypara = document.getElementById('closer');
mypara.addEventListener('click', function () {
  alert('Boo!');
}, false);
mypara.addEventListener(
  'click', console.log.bind(console), false);
```

As you can see, `addEventListeners()` is a method called on the node object and accepts the type of event as its first parameter and a function pointer as its second. You can use anonymous functions such as `function () { alert('Boo!'); }` or existing functions such as `console.log`. The listener functions you specify are called when the event happens and an argument is passed to them. This argument is an event object. If you run the preceding code and click on the last paragraph, you can see event objects being logged to the console. Clicking on an event object allows you to see its properties:



```
▼MouseEvent {webkitMovementY: 0, webkitMovementX: 0
   altKey: false
   bubbles: true
   button: 0
   cancelBubble: false
   cancelable: true
   charCode: 0
   clientX: 19
   clientY: 92
   clipboardData: undefined
   ctrlKey: false
   currentTarget: null
   dataTransfer: null
   defaultPrevented: false
   detail: 1
   eventPhase: 0
   fromElement: null
   keyCode: 0
   layerX: 19
   layerY: 92
   metaKey: false
   offsetX: 11
   offsetY: 8
   pageX: 19
   pageY: 92
   relatedTarget: null
   returnValue: true
   screenX: 254
   screenY: 186
   shiftKey: false
 ▶ srcElement: <p>
 ▶ target: <p>
   timeStamp: 1356056143153
 ▶ toElement: <p>
   type: "click"
 ▶ view: Window
   webkitMovementX: 0
   webkitMovementY: 0
   which: 1
   x: 19
   y: 92
 ▶ __proto__: MouseEvent
>
```

Capturing and bubbling

In the calls to `addEventListener()`, there was a third parameter, `false`. Let's see what is it for.

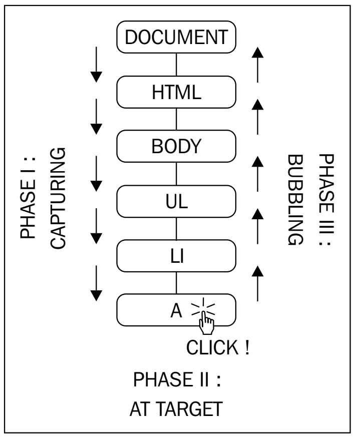
Say you have a link inside an unordered list as follows:

```
<body>
  <ul>
    <li><a href="http://phpied.com">my blog</a></li>
  </ul>
</body>
```

When you click on the link, you're actually also clicking on the list item `<li>`, the list `<ul>`, the `<body>`, and eventually the document as a whole. This is called event propagation. A click on a link can also be seen as click on the document. The process of propagating an event can be implemented in two ways:

- Event capturing: The click happens on the document first, then it propagates down to the body, the list, the list item, and finally to the link
- Event bubbling: The click happens on the link and then bubbles up to the document

DOM Level 2 events specification suggests that the events propagate in three phases, namely, capturing, at target, and bubbling. This means that the event propagates from the document to the link (target) and then bubbles back up to the document. The event objects have an `eventPhase` property, which reflects the current phase:



Historically, IE and Netscape (working on their own and without a standard to follow) implemented the exact opposites. IE implemented only bubbling, Netscape only capturing. Today, long after the DOM specification, modern browsers implement all three phases.

The practical implications related to the event propagation are as follows:

- The third parameter to `addEventListener()` specifies whether or not capturing should be used. In order to have your code more portable across browsers, it's better to always set this parameter to `false` and use bubbling only.
- You can stop the propagation of the event in your listeners so that it stops bubbling up and never reaches the document. To do this you can call the `stopPropagation()` method of the event object (there is an example in the next section).
- You can also use event delegation. If you have ten buttons inside a `<div>`, you can always attach ten event listeners, one for each button. But a smarter thing to do is to attach only one listener to the wrapping `<div>` and once the event happens, check which button was the target of the click.

As a side note, there is a way to use event capturing in old IEs too (using `setCapture()` and `releaseCapture()` methods) but only for mouse events. Capturing any other events (keystroke events for example) is not supported.

Stop propagation

Let's see an example of how you can stop the event from bubbling up. Going back to the test document, there is this piece of code:

```
<p id="closer">final</p>
```

Let's define a function that handles clicks on the paragraph:

```
function paraHandler() {
  alert('clicked paragraph');
}
```

Now, let's attach this function as a listener to the click event:

```
var para = document.getElementById('closer');
para.addEventListener('click', paraHandler, false);
```

Let's also attach listeners to the click event on the body, the document, and the browser window:

```
document.body.addEventListener('click', function () {
  alert('clicked body');
}, false);
document.addEventListener('click', function () {
  alert('clicked doc');
}, false);
window.addEventListener('click', function () {
  alert('clicked window');
}, false);
```

Note, that the DOM specifications don't say anything about events on the window. And why would they, since DOM deals with the document and not the browser. So browsers implement window events inconsistently.

Now, if you click on the paragraph, you'll see four alerts saying:

- clicked paragraph
- clicked body
- clicked doc
- clicked window

This illustrates how the same single click event propagates (bubbles up) from the target all the way up to the window.

The opposite of `addEventLister()` is `removeEventListener()` and it accepts exactly the same parameters. Let's remove the listener attached to the paragraph.

```
> para.removeEventListener('click', paraHandler, false);
```

If you try now, you'll see alerts only for the click event on the body, document, and window, but not on the paragraph.

Now, let's stop the propagation of the event. The function you add as a listener receives the event object as a parameter and you can call the `stopPropagation()` method of that event object as follows:

```
function paraHandler(e) {
  alert('clicked paragraph');
  e.stopPropagation();
}
```

Adding the modified listener is done as follows:

```
para.addEventListener('click', paraHandler, false);
```

Now, when you click on the paragraph you see only one alert because the event doesn't bubble up to the body, the document, or the window.

Note, that when you remove a listener, you have to pass a pointer to the same function you previously attached. Otherwise doing the following does not work because the second argument is a new function, not the same you passed when adding the event listener, even if the body is exactly the same:

```
document.body.removeEventListener('click',
  function () {
    alert('clicked body');
  },
false); //  does NOT remove the handler
```

Prevent default behavior

Some browser events have a predefined behavior. For example, clicking a link causes the browser to navigate to another page. You can attach listeners to clicks on a link and you can also disable the default behavior by calling the method `preventDefault()` on the event object.

Let's see how you can annoy your visitors by asking "Are you sure you want to follow this link?" every time they click a link. If the user clicks on **Cancel** (causing `confirm()` to return `false`), the `preventDefault()` method is called as follows:

```
// all links
var all_links = document.getElementsByTagName('a');
for (var i = 0; i < all_links.length; i++) { // loop all links
  all_links[i].addEventListener(
    'click',       // event type
    function (e) { // handler
      if (!confirm('Sure you want to follow this link?')) {
        e.preventDefault();
      }
    },
    false // don't use capturing
  );
}
```

Note, that not all events allow you to prevent the default behavior. Most do, but if you want to be sure, you can check the `cancellable` property of the event object.

Cross-browser event listeners

As you already know, most modern browsers almost fully implement the DOM Level 1 specification. However, the events were not standardized until DOM 2. As a result, there are quite a few differences in how IE before version 9 implements this functionality compared to modern browsers.

Check out an example that causes the `nodeName` of a clicked element (the target element) to be written to the console:

```
document.addEventListener('click', function (e) {
  console.log(e.target.nodeName);
}, false);
```

Now, let's take a look at how IE is different:

- In IE there's no `addEventListener()` method, although since IE Version 5 there is an equivalent `attachEvent()`. For earlier versions, your only choice is accessing the property (such as `onclick`) directly.
- `click` event becomes `onclick` when using `attachEvent()`.
- If you listen to events the old-fashioned way (for example, by setting a function value to the `onclick` property), when the callback function is invoked, it doesn't get an event object passed as a parameter. But, regardless of how you attach the listener in IE, there is always a global object `window.event` that points to the latest event.
- In IE the event object doesn't get a `target` attribute telling you the element on which the event fired, but it does have an equivalent property called `srcElement`.
- As mentioned before, event capturing doesn't apply to all events, so only bubbling should be used.
- There's no `stopPropagation()` method, but you can set the IE-only `cancelBubble` property to `true`.
- There's no `preventDefault()` method, but you can set the IE-only `returnValue` property to `false`.
- To stop listening to an event, instead of `removeEventListener()` in IE you'll need `detachEvent()`.

So, here's the revised version of the previous code that works across browsers:

```
function callback(evt) {
  // prep work
  evt = evt || window.event;
```

```
    var target = evt.target || evt.srcElement;

    // actual callback work
    console.log(target.nodeName);
  }

  //  start listening for click events
  if (document.addEventListener) { // Modern browsers
    document.addEventListener('click', callback, false);
  } else if (document.attachEvent) { // old IE
    document.attachEvent('onclick', callback);
  } else {
    document.onclick = callback; // ancient
  }
```

Types of events

Now you know how to handle cross-browser events. But all of the examples above used only click events. What other events are happening out there? As you can probably guess, different browsers provide different events. There is a subset of cross-browser events and some browser-specific ones. For a full list of events, you should consult the browser's documentation, but here's a selection of cross-browser events:

- Mouse events
  - mouseup, mousedown, click (the sequence is mousedown-up-click), dblclick
  - mouseover (mouse is over an element), mouseout (mouse was over an element but left it), mousemove
- Keyboard events
  - keydown, keypress, keyup (occur in this sequence)
- Loading/window events
  - load (an image or a page and all of its components are done loading), unload (user leaves the page), beforeunload (the script can provide the user with an option to stop the unload)
  - abort (user stops loading the page or an image in IE), error (a JavaScript error, also when an image cannot be loaded in IE)
  - resize (the browser window is resized), scroll (the page is scrolled), contextmenu (the right-click menu appears)

- Form events
    - focus (enter a form field), blur (leave the form field)
    - change (leave a field after the value has changed), select (select text in a text field)
    - reset (wipe out all user input), submit (send the form)

Additionally, modern browsers provide drag events (dragstart, dragend, drop, and others) and touch devices provide touchstart, touchmove, and touchend.

This concludes the discussion of events. Refer to the exercise section at the end of this chapter for a little challenge of creating your own event utility to handle cross-browser events.

XMLHttpRequest

`XMLHttpRequest()` is an object (a constructor function) that allows you to send HTTP requests from JavaScript. Historically, `XMLHttpRequest` (or XHR for short) was introduced in IE and was implemented as an ActiveX object. Starting with IE7 it's a native browser object, the same way as it's in the other browsers. The common implementation of this object across browsers gave birth to the so-called Ajax applications, where it's no longer necessary to refresh the whole page every time you need new content. With JavaScript, you can make an HTTP request to the server, get the response, and update only a part of the page. This way you can build much more responsive and desktop-like web pages.

Ajax stands for **Asynchronous JavaScript and XML**.

- Asynchronous because after sending an HTTP request your code doesn't need to wait for the response, but it can do other stuff and be notified (through an event) when the response arrives.
- JavaScript because it's obvious that XHR objects are created with JavaScript.
- XML because initially developers were making HTTP requests for XML documents and were using the data contained in them to update the page. This is no longer a common practice, though, as you can request data in plain text, in the much more convenient JSON format, or simply as HTML ready to be inserted into the page.

There are two steps to using the `XMLHttpRequest`:

- **Send the request**: This includes creating an `XMLHttpRequest` object and attaching an event listener
- **Process the response**: Your event listener gets notified that the response has arrived and your code gets busy doing something amazing with the response

Sending the request

In order to create an object you simply use the following code (let's deal with browser inconsistencies in just a bit):

```
var xhr = new XMLHttpRequest();
```

The next thing is to attach an event listener to the `readystatechange` event fired by the object:

```
xhr.onreadystatechange = myCallback;
```

Then, you need to call the `open()` method, as follows:

```
xhr.open('GET', 'somefile.txt', true);
```

The first parameter specifies the type of HTTP request (`GET`, `POST`, `HEAD`, and so on). `GET` and `POST` are the most common. Use `GET` when you don't need to send much data with the request and your request doesn't modify (write) data on the server, otherwise use `POST`. The second parameter is the URL you are requesting. In this example, it's the text file `somefile.txt` located in the same directory as the page. The last parameter is a boolean specifying whether the request is asynchronous (`true`, always prefer this) or not (`false`, blocks all the JavaScript execution and waits until the response arrives).

The last step is to fire off the request which is done as follows:

```
xhr.send('');
```

The method `send()` accepts any data you want to send with the request. For `GET` requests, this is an empty string, because the data is in the URL. For `POST` request, it's a query string in the form `key=value&key2=value2`.

At this point, the request is sent and your code (and the user) can move on to other tasks. The callback function `myCallback` will be invoked when the response comes back from the server.

Processing the response

A listener is attached to the `readystatechange` event. So what exactly is the ready state and how does it change?

There is a property of the XHR object called `readyState`. Every time it changes, the `readystatechange` event fires. The possible values of the `readyState` property are as follows:

- 0-uninitialized
- 1-loading
- 2-loaded
- 3-interactive
- 4-complete

When `readyState` gets the value of 4, it means the response is back and ready to be processed. In `myCallback` after you make sure `readyState` is 4, the other thing to check is the status code of the HTTP request. You might have requested a non-existing URL for example and get a 404 (File not found) status code. The interesting code is the 200 (OK) code, so `myCallback` should check for this value. The status code is available in the `status` property of the XHR object.

Once `xhr.readyState` is 4 and `xhr.status` is 200, you can access the contents of the requested URL using the `xhr.responseText` property. Let's see how `myCallback` could be implemented to simply `alert()` the contents of the requested URL:

```
function myCallback() {

  if (xhr.readyState < 4) {
    return; // not ready yet
  }

  if (xhr.status !== 200) {
    alert('Error!'); // the HTTP status code is not OK
    return;
  }

  //  all is fine, do the work
  alert(xhr.responseText);
}
```

Once you've received the new content you requested, you can add it to the page, or use it for some calculations, or for any other purpose you find suitable.

Overall, this two-step process (send request and process response) is the core of the whole XHR/Ajax functionality. Now that you know the basics, you can move on to building the next Gmail. Oh yes, let's have a look at some minor browser inconsistencies.

Creating XMLHttpRequest objects in IE prior to Version 7

In Internet Explorer prior to version 7, the `XMLHttpRequest` object was an ActiveX object, so creating an XHR instance is a little different. It goes like the following:

```
var xhr = new ActiveXObject('MSXML2.XMLHTTP.3.0');
```

MSXML2.XMLHTTP.3.0 is the identifier of the object you want to create. There are several versions of the `XMLHttpRequest` object and if your page visitor doesn't have the latest one installed, you can try two older ones, before you give up.

For a fully-cross-browser solution, you should first test to see if the user's browser supports `XMLHttpRequest` as a native object, and if not, try the IE way. Therefore, the whole process of creating an XHR instance could be like this:

```
var ids = ['MSXML2.XMLHTTP.3.0',
           'MSXML2.XMLHTTP',
           'Microsoft.XMLHTTP'];

var xhr;
if (XMLHttpRequest) {
  xhr = new XMLHttpRequest();
} else {
  // IE: try to find an ActiveX object to use
  for (var i = 0; i < ids.length; i++) {
    try {
      xhr = new ActiveXObject(ids[i]);
      break;
    } catch (e) {}
  }
}
```

What is this doing? The array `ids` contains a list of ActiveX program IDs to try. The variable `xhr` points to the new XHR object. The code first checks to see if `XMLHttpRequest` exists. If so, this means that the browser supports `XMLHttpRequest()` natively (so the browser is relatively modern). If it is not, the code loops through `ids` trying to create an object. `catch(e)` quietly ignores failures and the loop continues. As soon as an `xhr` object is created, you `break` out of the loop.

As you can see, this is quite a bit of code so it's best to abstract it into a function. Actually, one of the exercises at the end of the chapter prompts you to create your own Ajax utility.

A is for Asynchronous

Now you know how to create an XHR object, give it a URL and handle the response to the request. What happens when you send two requests asynchronously? What if the response to the second request comes before the first?

In the example above, the XHR object was global and `myCallback` was relying on the presence of this global object in order to access its `readyState`, `status`, and `responseText` properties. Another way, which prevents you from relying on global variables, is to wrap the callback in a closure. Let's see how:

```
var xhr = new XMLHttpRequest();

xhr.onreadystatechange = (function (myxhr) {
  return function () {
    myCallback(myxhr);
  };
}(xhr));

xhr.open('GET', 'somefile.txt', true);
xhr.send('');
```

In this case `myCallback()` receives the XHR object as a parameter and is not going to look for it in the global space. This also means that at the time the response is received, the original `xhr` might have been reused for a second request. The closure keeps pointing to the original object.

X is for XML

Although these days JSON (discussed in the next chapter) is preferred over XML as a data transfer format, XML is still an option. In addition to the `responseText` property, the XHR objects also have another property called `responseXML`. When you send an HTTP request for an XML document, `responseXML` points to an XML DOM document object. To work with this document, you can use all of the core DOM methods discussed previously in this chapter, such as `getElementsByTagName()`, `getElementById()`, and so on.

An example

Let's wrap up the different XHR topics with an example. You can visit the page located at `http://www.phpied.com/files/jsoop/xhr.html` to work on the example yourself:

The main page, `xhr.html`, is a simple static page that contains nothing but three `<div>` tags.

```
<div id="text">Text will be here</div>
<div id="html">HTML will be here</div>
<div id="xml">XML will be here</div>
```

Using the console, you can write code that requests three files and loads their respective contents into each `<div>`.

The three files to load are:

- `content.txt`: a simple text file containing the text "I am a text file"
- `content.html`: a file containing HTML code `"I am <strong>formatted</strong> <em>HTML</em>"`
- `content.xml`: an XML file, containing the following code:

  ```
  <?xml version="1.0" ?>
  <root>
      I'm XML data.
  </root>
  ```

All of the files are stored in the same directory as `xhr.html`.

For security reasons you can only use the original `XMLHttpRequest` to request files that are on the same domain. However, modern browsers support XHR2 which lets you make cross-domain requests, provided that the appropriate Access-Control-Allow-Origin HTTP header is in place.

First, let's create a function to abstract the request/response part:

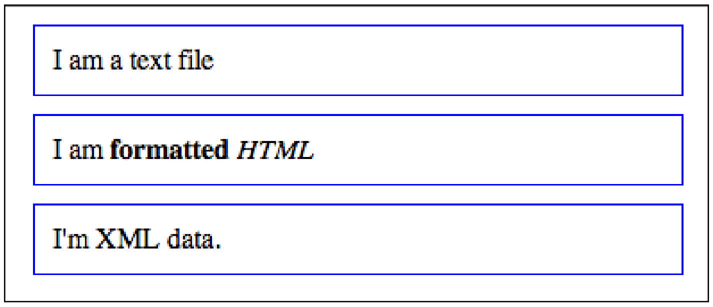
```
function request(url, callback) {
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = (function (myxhr) {
    return function () {
      if (myxhr.readyState === 4 && myxhr.status === 200) {
        callback(myxhr);
      }
    };
```

```
    }(xhr));
    xhr.open('GET', url, true);
    xhr.send('');
  }
```

This function accepts a URL to request and a callback function to call once the response arrives. Let's call the function three times, once for each file, as follows:

```
  request(
    'http://www.phpied.com/files/jsoop/content.txt',
    function (o) {
      document.getElementById('text').innerHTML =
        o.responseText;
    }
  );
  request(
    'http://www.phpied.com/files/jsoop/content.html',
    function (o) {
      document.getElementById('html').innerHTML =
        o.responseText;
    }
  );
  request(
    'http://www.phpied.com/files/jsoop/content.xml',
    function (o) {
      document.getElementById('xml').innerHTML =
        o.responseXML
         .getElementsByTagName('root')[0]
         .firstChild
         .nodeValue;
    }
  );
```

The callback functions are defined inline. The first two are identical. They just replace the HTML of the corresponding `<div>` with the contents of the requested file. The third one is a little different as it deals with the XML document. First, you access the XML DOM object as `o.responseXML`. Then, using `getElementsByTagName()` you get a list of all `<root>` tags (there is only one). The `firstChild` of `<root>` is a text node and `nodeValue` is the text contained in it ("I'm XML data"). Then just replace the HTML of `<div id="xml">` with the new content. The result is shown on the following screenshot:


I am a text file

I am **formatted** *HTML*

I'm XML data.

When working with the XML document, you can also use `o.responseXML.documentElement` to get to the `<root>` element, instead of `o.responseXML.getElementsByTagName('root')[0]`. Remember that `documentElement` gives you the root node of an XML document. The root in HTML documents is always the `<html>` tag.

Summary

You learned quite a bit in this chapter. You have learned some cross-browser BOM (Browser Object Model) objects:

- Properties of the global `window` object such as `navigator`, `location`, `history`, `frames`, `screen`
- Methods such as `setInterval()` and `setTimeout()`; `alert()`, `confirm()` and `prompt()`; `moveTo/By()` and `resizeTo/By()`

Then you learned about the DOM (Document Object Model), an API to represent an HTML (or XML) document as a tree structure where each tag or text is a node on the tree. You also learned how to do the following actions:

- Access nodes
    - Using parent/child relationship properties `parentNode`, `childNodes`, `firstChild`, `lastChild`, `nextSibling`, and `previousSibling`
    - Using `getElementsById()`, `getElementsByTagName()`, `getElementsByName()`, and `querySelectorAll()`
- Modify nodes:
    - Using `innerHTML` or `innerText/textContent`
    - Using `nodeValue` or `setAttribute()` or just using attributes as object properties

- Remove nodes with `removeChild()` or `replaceChild()`
- And add new ones with `appendChild()`, `cloneNode()`, and `insertBefore()`

You also learned some DOM0 (prestandardization) properties, ported to DOM Level 1:

- Collections such as `document.forms`, `images`, `links`, `anchors`, `applets`. Using these are discouraged as DOM1 has the much more flexible method `getElementsByTagName()`.
- `document.body` which gives you convenient access to `<body>`.
- `document.title`, `cookie`, `referrer`, and `domain`.

Next, you learned about how the browser broadcasts events that you can listen to. It's not straightforward to do this in a cross-browser manner, but it's possible. Events bubble up, so you can use event delegation to listen to events more globally. You can also stop the propagation of events and interfere with the default browser behavior.

Finally, you learned about the `XMLHttpRequest` object that allows you to build responsive web pages that do the following tasks:

- Make HTTP requests to the server to get pieces of data
- Process the response to update portions of the page

Exercises

In the previous chapters, the solutions to the exercises could be found in the text of the chapter. This time, some of the exercises require you to do some more reading (or experimentation) outside this book.

1. BOM: As a BOM exercise, try coding something wrong, obtrusive, user-unfriendly, and all in all, very Web 1.0, the shaking browser window. Try implementing code that opens a 200 x 200 pop up window and then resizes it slowly and gradually to 400 x 400. Next, move the window around as if there's an earthquake. All you'll need is one of the `move*()` functions, one or more calls to `setInterval()`, and maybe one to `setTimeout()/clearInterval()` to stop the whole thing. Or here's an easier one, print the current date/time in the `document.title` and update it every second, like a clock.
2. DOM:
   - Implement `walkDOM()` differently. Also make it accept a callback function instead of hard coding `console.log()`

- Removing content with `innerHTML` is easy (`document.body.innerHTML = ''`), but not always best. The problem will be when there are event listeners attached to the removed elements, they won't be removed in IE causing the browser to leak memory, because it stores references to something that doesn't exist. Implement a general-purpose function that deletes DOM nodes, but removes any event listeners first. You can loop through the attributes of a node and check if the value is a function. If it is, it's most likely an attribute like `onclick`. You need to set it to `null` before removing the element from the tree.
- Create a function called `include()` that includes external scripts on demand. This means you need to create a new `<script>` tag dynamically, set its `src` attribute and append to the document's `<head>`. Test by using the following code:

  ```
  > include('somescript.js');
  ```

3. Events:

   Create an event utility (object) called `myevent` which has the following methods working cross-browser:

   - `addListener(element, event_name, callback)` where `element` could also be an array of elements
   - `removeListener(element, event_name, callback)`
   - `getEvent(event)` just to check for a `window.event` for older versions of IE
   - `getTarget(event)`
   - `stopPropagation(event)`
   - `preventDefault(event)`

   Usage example:

   ```
   function myCallback(e) {
     e = myevent.getEvent(e);
     alert(myevent.getTarget(e).href);
     myevent.stopPropagation(e);
     myevent.preventDefault(e);
   }
   myevent.addListener(document.links, 'click', myCallback);
   ```

   The result of the example code should be that all of the links in the document lead nowhere but only alert the `href` attribute.

Create an absolutely positioned `<div>`, say at `x = 100px`, `y = 100px`. Write the code to be able to move the div around the page using the arrow keys or the keys *J* (left), *K* (right), *M* (down), and *I* (up). Reuse your own event utility from 3.1.

4. XMLHttpRequest

   Create your own XHR utility (object) called `ajax`. For example, have a look at the following code:

```
function myCallback(xhr) {
  alert(xhr.responseText);
}
ajax.request('somefile.txt', 'get', myCallback);
ajax.request('script.php', 'post', myCallback,
             'first=John&last=Smith');
```

8
Coding and Design Patterns

Now that you know all about the objects in JavaScript, you've mastered prototypes and inheritance, and you have seen some practical examples of using browser-specific objects, let's move forward, or rather move a level up. Let's have a look at some common JavaScript patterns.

But first, what's pattern? In short, a pattern is a good solution to a common problem.

Sometimes when you're facing a new programming problem, you may recognize right away that you've previously solved another, suspiciously similar problem. In such cases, it's worth isolating this class of problems and searching for a common solution. A pattern is a proven and reusable solution (or an approach to a solution) to a class of problems.

There are cases where a pattern is nothing more than an idea or a name. Sometimes just using a name helps you think more clearly about a problem. Also, when working with other developers in a team, it's much easier to communicate when everybody uses the same terminology to discuss a problem or a solution.

Other times you may come across a unique problem that doesn't look like anything you've seen before and doesn't readily fit into a known pattern. Blindly applying a pattern just for the sake of using a pattern is not a good idea. It's preferable to not use any known pattern than to try to tweak your problem so that it fits an existing solution.

This chapter talks about two types of patterns:

- **Coding patterns**: These are mostly JavaScript-specific best practices
- **Design patterns**: These are language-independent patterns, popularized by the famous "Gang of Four" book

Coding patterns

Let's start with some patterns that reflect JavaScript's unique features. Some patterns aim to help you organize your code (for example, namespacing), others are related to improving performance (such as lazy definitions and init-time branching), and some make up for missing features such as private properties. The patterns discussed in this section include:

- Separating behavior
- Namespaces
- Init-time branching
- Lazy definition
- Configuration objects
- Private variables and methods
- Privileged methods
- Private functions as public methods
- Immediate functions
- Chaining
- JSON

Separating behavior

As discussed previously, the three building blocks of a web page are as follows:

- Content (HTML)
- Presentation (CSS)
- Behavior (JavaScript)

Content

HTML is the content of the web page, the actual text. Ideally, the content should be marked up using the least amount of HTML tags that sufficiently describe the semantic meaning of that content. For example, if you're working on a navigation menu it's a good idea to use `<ul>` and `<li>` since a navigation menu is in essence just a list of links.

Your content (HTML) should be free from any formatting elements. Visual formatting belongs to the presentation layer and should be achieved through the use of **Cascading Style Sheets** (**CSS**). This means the following:

- The `style` attribute of HTML tags should not be used, if possible.
- Presentational HTML tags such as `<font>` should not be used at all.
- Tags should be used for their semantic meaning, not because of how browsers render them by default. For instance, developers sometimes use a `<div>` tag where a `<p>` would be more appropriate. It's also favorable to use `<strong>` and `<em>` instead of `<b>` and `<i>` as the latter describe the visual presentation rather than the meaning.

Presentation

A good approach to keep presentation out of the content is to reset, or nullify all browser defaults. For example, using `reset.css` from the Yahoo! UI library. This way the browser's default rendering won't distract you from consciously thinking about the proper semantic tags to use.

Behavior

The third component of a web page is the behavior. Behavior should be kept separate from both the content and the presentation. Behavior is usually added by using JavaScript that is isolated to `<script>` tags, and preferably contained in external files. This means not using any inline attributes such as `onclick`, `onmouseover`, and so on. Instead, you can use the `addEventListener`/`attachEvent` methods from the previous chapter.

The best strategy for separating behavior from content is as follows:

- Minimize the number of `<script>` tags
- Avoid inline event handlers
- Do not use CSS expressions
- Dynamically add markup that has no purpose if JavaScript is disabled by the user
- Towards the end of your content when you are ready to close the `<body>` tag, insert a single `external.js` file

Example of separating behavior

Let's say you have a search form on a page and you want to validate the form with JavaScript. So, you go ahead and keep the form tags free from any JavaScript, and then immediately before the closing the `</body>` tag you insert a `<script>` tag which links to an external file as follows:

```
<body>
  <form id="myform" method="post" action="server.php">
  <fieldset>
    <legend>Search</legend>
    <input
      name="search"
      id="search"
      type="text"
    />
    <input type="submit" />
    </fieldset>
  </form>
  <script src="behaviors.js"></script>
</body>
```

In `behaviors.js` you attach an event listener to the submit event. In your listener, you check to see if the text input field was left blank and if so, stop the form from being submitted. This way you save a round trip between the server and the client and make the application immediately responsive.

The content of `behaviors.js` is given in the following code. It assumes that you've created your `myevent` utility from the exercise at the end of the previous chapter:

```
// init
myevent.addListener('myform', 'submit', function (e) {
  // no need to propagate further
  e = myevent.getEvent(e);
  myevent.stopPropagation(e);
  // validate
  var el = document.getElementById('search');
  if (!el.value) { // too bad, field is empty
    myevent.preventDefault(e); // prevent the form submission
    alert('Please enter a search string');
  }
});
```

Asynchronous JavaScript loading

You noticed how the script was loaded at the end of the HTML right before closing the body. The reason is that JavaScript blocks the DOM construction of the page and in some browsers even the downloads of the other components that follow. By moving the scripts to the bottom of the page you ensure the script is out of the way and when it arrives, it simply enhances the already usable page.

Another way to prevent external JavaScript files from blocking the page is to load them asynchronously. This way you can start loading them earlier. HTML5 has the `defer` attribute for this purpose:

```
<script defer src="behaviors.js"></script>
```

Unfortunately, the `defer` attribute is not supported by older browsers, but luckily, there is a solution that works across browsers, old and new. The solution is to create a script node dynamically and append it to the DOM. In other words you use a bit of inline JavaScript to load the external JavaScript file. You can have this script loader snippet at the top of your document so that the download has an early start:

```
...
<head>
(function () {
  var s = document.createElement('script');
  s.src = 'behaviors.js';
  document.getElementsByTagName('head')[0].appendChild(s);
}());
</head>
...
```

Namespaces

Global variables should be avoided in order to reduce the possibility of variable naming collisions. You can minimize the number of globals by namespacing your variables and functions. The idea is simple, you create only one global object and all your other variables and functions become properties of that object.

An Object as a namespace

Let's create a global object called `MYAPP`:

```
// global namespace
var MYAPP = MYAPP || {};
```

Now, instead of having a global `myevent` utility (from the previous chapter), you can have it as an `event` property of the `MYAPP` object as follows:

```
// sub-object
MYAPP.event = {};
```

Adding the methods to the `event` utility is still the same:

```
// object together with the method declarations
MYAPP.event = {
  addListener: function (el, type, fn) {
    // .. do the thing
  },
  removeListener: function (el, type, fn) {
    // ...
  },
  getEvent: function (e) {
    // ...
  }
  // ... other methods or properties
};
```

Namespaced constructors

Using a namespace doesn't prevent you from creating constructor functions. Here is how you can have a DOM utility that has an `Element` constructor, which allows you to create DOM elements easier:

```
MYAPP.dom = {};
MYAPP.dom.Element = function (type, properties) {
  var tmp = document.createElement(type);
  for (var i in properties) {
    if (properties.hasOwnProperty(i)) {
      tmp.setAttribute(i, properties[i]);
    }
  }
  return tmp;
};
```

Similarly, you can have a `Text` constructor to create text nodes:

```
MYAPP.dom.Text = function (txt) {
  return document.createTextNode(txt);
};
```

Using the constructors to create a link at the bottom of a page can be done as follows:

```
var link = new MYAPP.dom.Element('a',
  {href: 'http://phpied.com', target: '_blank'});
var text = new MYAPP.dom.Text('click me');
link.appendChild(text);
document.body.appendChild(link);
```

A namespace() method

You can create a namespace utility that makes your life easier so that you can use more convenient syntax:

```
MYAPP.namespace('dom.style');
```

Instead of the more verbose syntax as follows:

```
MYAPP.dom = {};
MYAPP.dom.style = {};
```

Here's how you can create such a `namespace()` method. First, you create an array by splitting the input string using the period (.) as a separator. Then, for every element in the new array, you add a property to your global object, if one doesn't already exist as follows:

```
var MYAPP = {};
MYAPP.namespace = function (name) {
  var parts = name.split('.');
  var current = MYAPP;
  for (var i = 0; i < parts.length; i++) {
    if (!current[parts[i]]) {
      current[parts[i]] = {};
    }
    current = current[parts[i]];
  }
};
```

Testing the new method is done as follows:

```
MYAPP.namespace('event');
MYAPP.namespace('dom.style');
```

The result of the preceding code is the same as if you did the following:

```
var MYAPP = {
  event: {},
  dom: {
    style: {}
  }
};
```

Init-time branching

In the previous chapter you noticed that sometimes different browsers have different implementations for the same or similar functionalities. In such cases, you need to branch your code depending on what's supported by the browser currently executing your script. Depending on your program this branching can happen far too often and, as a result, may slow down the script execution.

You can mitigate this problem by branching some parts of the code during initialization, when the script loads, rather than during runtime. Building upon the ability to define functions dynamically, you can branch and define the same function with a different body depending on the browser. Let's see how.

First, let's define a namespace and placeholder method for the `event` utility:

```
var MYAPP = {};
MYAPP.event = {
  addListener: null,
  removeListener: null
};
```

At this point, the methods to add or remove a listener are not implemented. Based on the results from feature sniffing, these methods can be defined differently as follows:

```
if (window.addEventListener) {
  MYAPP.event.addListener = function (el, type, fn) {
    el.addEventListener(type, fn, false);
  };
  MYAPP.event.removeListener = function (el, type, fn) {
    el.removeEventListener(type, fn, false);
  };
} else if (document.attachEvent) { // IE
  MYAPP.event.addListener = function (el, type, fn) {
    el.attachEvent('on' + type, fn);
  };
  MYAPP.event.removeListener = function (el, type, fn) {
    el.detachEvent('on' + type, fn);
  };
} else { // older browsers
  MYAPP.event.addListener = function (el, type, fn) {
    el['on' + type] = fn;
  };
  MYAPP.event.removeListener = function (el, type) {
    el['on' + type] = null;
  };
}
```

After this script executes, you have the `addListener()` and `removeListener()` methods defined in a browser-dependent way. Now, every time you invoke one of these methods there's no more feature-sniffing and it results in less work and faster execution.

One thing to watch out for when sniffing features is not to assume too much after checking for one feature. In the previous example this rule is broken because the code only checks for `addEventListener` support but then defines both `addListener()` and `removeListener()`. In this case it's probably safe to assume that if a browser implements `addEventListener()` it also implements `removeEventListener()`. But, imagine what happens if a browser implements `stopPropagation()` but not `preventDefault()` and you haven't checked for these individually. You have assumed that because `addEventListener()` is not defined, the browser must be an old IE and write your code using your knowledge and assumptions of how IE works. Remember that all of your knowledge is based on the way a certain browser works today, but not necessarily the way it will work tomorrow. So to avoid many rewrites of your code as new browser versions are shipped, it's best to individually check for features you intend to use and don't generalize on what a certain browser supports.

Lazy definition

The lazy definition pattern is similar to the previous init-time branching pattern. The difference is that the branching happens only when the function is called for the first time. When the function is called, it redefines itself with the best implementation. Unlike the init-time branching where the if happens once, during loading, here it may not happen at all in cases when the function is never called. The lazy definition also makes the initialization process lighter as there's no init-time branching work to be done.

Let's see an example that illustrates this via the definition of an `addListener()` function. The function is first defined with a generic body. It checks which functionality is supported by the browser when it's called for the first time and then redefines itself using the most suitable implementation. At the end of the first call, the function calls itself so that the actual event attaching is performed. The next time you call the same function it will be defined with its new body and be ready for use, so no further branching is necessary. Following is the code snippet:

```
var MYAPP = {};
MYAPP.myevent = {
  addListener: function (el, type, fn) {
    if (el.addEventListener) {
      MYAPP.myevent.addListener = function (el, type, fn) {
        el.addEventListener(type, fn, false);
      };
    } else if (el.attachEvent) {
      MYAPP.myevent.addListener = function (el, type, fn) {
        el.attachEvent('on' + type, fn);
      };
    } else {
      MYAPP.myevent.addListener = function (el, type, fn) {
        el['on' + type] = fn;
      };
    }
    MYAPP.myevent.addListener(el, type, fn);
  }
};
```

Configuration object

This pattern is convenient when you have a function or method that accepts a lot of optional parameters. It's up to you to decide how many constitutes a lot. But generally, a function with more than three parameters is not convenient to call because you have to remember the order of the parameters, and it is even more inconvenient when some of the parameters are optional.

Instead of having many parameters, you can use one parameter and make it an object. The properties of the object are the actual parameters. This is suitable for passing configuration options because these tend to be numerous and optional (with smart defaults). The beauty of using a single object as opposed to multiple parameters is described as follows:

- The order doesn't matter
- You can easily skip parameters that you don't want to set

- It's easy to add more optional configuration attributes
- It makes the code more readable because the configuration object's properties are present in the calling code along with their names

Imagine you have some sort of UI widget constructor you use to create fancy buttons. It accepts the text to put inside the button (the `value` attribute of the `<input>` tag) and an optional parameter of the `type` of button. For simplicity let's say the fancy button takes the same configuration as a regular button. Have a look at the following code:

```
// a constructor that creates buttons
MYAPP.dom.FancyButton = function (text, type) {
  var b = document.createElement('input');
  b.type = type || 'submit';
  b.value = text;
  return b;
};
```

Using the constructor is simple; you just give it a string. Then you can add the new button to the body of the document:

```
document.body.appendChild(
  new MYAPP.dom.FancyButton('puuush')
);
```

This is all well and works fine, but then you decide you also want to be able to set some of the style properties of the button such as colors and fonts. You can end up with a definition like the following:

```
MYAPP.dom.FancyButton =
  function (text, type, color, border, font) {
  // ...
};
```

Now, using the constructor can become a little inconvenient, especially when you want to set the third and fifth parameter, but not the second or the fourth:

```
new MYAPP.dom.FancyButton(
  'puuush', null, 'white', null, 'Arial');
```

A better approach is to use one config object parameter for all the settings. The function definition can become something like the following:

```
MYAPP.dom.FancyButton = function (text, conf) {
  var type = conf.type || 'submit';
  var font = conf.font || 'Verdana';
  // ...
};
```

Using the constructor is given as follows:

```
var config = {
  font: 'Arial, Verdana, sans-serif',
  color: 'white'
};
new MYAPP.dom.FancyButton('puuush', config);
```

Another usage example is as follows:

```
document.body.appendChild(
  new MYAPP.dom.FancyButton('dude', {color: 'red'})
);
```

As you can see, it's easy to set only some of the parameters and to switch around their order. In addition, the code is friendlier and easier to understand when you see the names of the parameters at the same place where you call the method.

A drawback of this pattern is the same as its strength. It's trivial to keep adding more parameters, which means trivial to abuse the technique. Once you have an excuse to add to this free-for-all bag of properties, you will find it tempting to keep adding some that are not entirely optional or some that are dependent on other properties.

As a rule of thumb, all these properties should be independent and optional. If you have to check all possible combinations inside your function ("oh, A is set, but A is only used if B is also set") this is a recipe for a large function body, which quickly becomes confusing and difficult, if not impossible, to test, because of all the combinations.

Private properties and methods

JavaScript doesn't have the notion of access modifiers, which set the privileges of the properties in an object. Other languages often have access modifiers such as:

- Public—all users of an object can access these properties (or methods)
- Private—only the object itself can access these properties
- Protected—only objects inheriting the object in question can access these properties

JavaScript doesn't have a special syntax to denote private properties or methods, but as discussed in *Chapter 3*, *Functions*, you can use local variables and methods inside a function and achieve the same level of protection.

Continuing with the example of the `FancyButton` constructor, you can have a local variable styles which contains all the defaults, and a local `setStyle()` function. These are invisible to the code outside of the constructor. Here's how `FancyButton` can make use of the local private properties:

```
var MYAPP = {};
MYAPP.dom = {};
MYAPP.dom.FancyButton = function (text, conf) {
  var styles = {
    font: 'Verdana',
    border: '1px solid black',
    color: 'black',
    background: 'grey'
  };
  function setStyles(b) {
    var i;
    for (i in styles) {
      if (styles.hasOwnProperty(i)) {
        b.style[i] = conf[i] || styles[i];
      }
    }
  }
  conf = conf || {};
  var b = document.createElement('input');
  b.type = conf.type || 'submit';
  b.value = text;
  setStyles(b);
  return b;
};
```

In this implementation, `styles` is a private property and `setStyle()` is a private method. The constructor uses them internally (and they can access anything inside the constructor), but they are not available to code outside of the function.

Privileged methods

Privileged methods (this term was coined by Douglas Crockford) are normal public methods that can access private methods or properties. They can act like a bridge in making some of the private functionality accessible but in a controlled manner, wrapped in a privileged method.

Private functions as public methods

Let us say you've defined a function that you absolutely need to keep intact, so you make it private. But, you also want to provide access to the same function so that outside code can also benefit from it. In this case, you can assign the private function to a publicly available property.

Let's define `_setStyle()` and `_getStyle()` as private functions, but then assign them to the public `setStyle()` and `getStyle()`:

```
var MYAPP = {};
MYAPP.dom = (function () {
  var _setStyle = function (el, prop, value) {
    console.log('setStyle');
  };
  var _getStyle = function (el, prop) {
    console.log('getStyle');
  };
  return {
    setStyle: _setStyle,
    getStyle: _getStyle,
    yetAnother: _setStyle
  };
}());
```

Now, when you call `MYAPP.dom.setStyle()`, it invokes the private `_setStyle()` function. You can also overwrite `setStyle()` from the outside:

```
MYAPP.dom.setStyle = function () {alert('b');};
```

Now, the result is as follows:

- `MYAPP.dom.setStyle` points to the new function
- `MYAPP.dom.yetAnother` still points to `_setStyle()`
- `_setStyle()` is always available when any other internal code relies on it to be working as intended, regardless of the outside code

When you expose something private, keep in mind that objects (and functions and arrays are objects too) are passed by reference and, therefore, can be modified from the outside.

Immediate functions

Another pattern that helps you keep the global namespace clean is to wrap your code in an anonymous function and execute that function immediately. This way any variables inside the function are local (as long as you use the `var` statement) and are destroyed when the function returns, if they aren't part of a closure. This pattern was discussed in more detail in *Chapter 3*, *Functions*. Have a look at the following code:

```
(function () {
  // code goes here...
}());
```

This pattern is especially suitable for on-off initialization tasks performed when the script loads.

The immediate (self-executing) function pattern can be extended to create and return objects. If the creation of these objects is more complicated and involves some initialization work, then you can do this in the first part of the self-executable function and return a single object, which can access and benefit from any private properties in the top portion as follows:

```
var MYAPP = {};
MYAPP.dom = (function () {
  // initialization code...
  function _private() {
    // ...
  }
  return {
    getStyle: function (el, prop) {
      console.log('getStyle');
      _private();
    },
    setStyle: function (el, prop, value) {
      console.log('setStyle');
    }
  };
}());
```

Modules

Combining several of the previous patterns, gives you a new pattern, commonly referred to as a module pattern. The concept of modules in programming is convenient as it allows you to code separate pieces or libraries and combine them as needed just like pieces of a puzzle.

Two notable facts beyond the scope of this chapter

JavaScript doesn't have a built-in concept of modules, although this is planned for the future via `export` and `import` declarations. There is also the module specification from `http://www.commonjs.org`, which defines a `require()` function and an `exports` object.

The module pattern includes:

- Namespaces to reduce naming conflicts among modules
- An immediate function to provide a private scope and initialization
- Private properties and methods
- Returning an object that has the public API of the module as follows:

```
namespace('MYAPP.module.amazing');

MYAPP.module.amazing = (function () {

  // short names for dependencies
  var another = MYAPP.module.another;

  // local/private variables
  var i, j;

  // private functions
  function hidden() {}

  // public API
  return {
    hi: function () {
      return "hello";
    }
  };
}());
```

And using the following module:

```
MYAPP.module.amazing.hi(); // "hello"
```

Chaining

Chaining is a pattern that allows you to invoke multiple methods on one line as if the methods are the links in a chain. This is convenient when calling several related methods. You invoke the next method on the result of the previous without the use of an intermediate variable.

Say you've created a constructor that helps you work with DOM elements. The code to create a new `<span>` and add it to the `<body>` could look something like the following:

```
var obj = new MYAPP.dom.Element('span');
obj.setText('hello');
obj.setStyle('color', 'red');
obj.setStyle('font', 'Verdana');
document.body.appendChild(obj);
```

As you know, constructors return the object referred to as `this` that they create. You can make your methods such as `setText()` and `setStyle()` also return `this`, which allows you to call the next method on the instance returned by the previous one. This way you can chain method calls:

```
var obj = new MYAPP.dom.Element('span');
obj.setText('hello')
   .setStyle('color', 'red')
   .setStyle('font', 'Verdana');
document.body.appendChild(obj);
```

You don't even need the `obj` variable if you don't plan on using it after the new element has been added to the tree, so the code looks like the following:

```
document.body.appendChild(
  new MYAPP.dom.Element('span')
    .setText('hello')
    .setStyle('color', 'red')
    .setStyle('font', 'Verdana')
);
```

A drawback of this pattern is that it makes it a little harder to debug when an error occurs somewhere in a long chain and you don't know which link is to blame because they are all on the same line.

JSON

Let's wrap up the coding patterns section of this chapter with a few words about JSON. JSON is not technically a coding pattern, but you can say that using JSON is a good pattern.

JSON is a popular lightweight format for exchanging data. It's often preferred over XML when using `XMLHttpRequest()` to retrieve data from the server. **JSON** stands for **JavaScript Object Notation** and there's nothing specifically interesting about it other than the fact that it's extremely convenient. The JSON format consists of data defined using object, and array literals. Here is an example of a JSON string that your server could respond with after an XHR request:

```
{
  'name':   'Stoyan',
  'family': 'Stefanov',
  'books':  ['OOJS', 'JSPatterns', 'JS4PHP']
}
```

An XML equivalent of this would be something like the following:

```
<?xml version="1.1" encoding="iso-8859-1"?>
<response>
  <name>Stoyan</name>
  <family>Stefanov</family>
  <books>
    <book>OOJS</book>
    <book>JSPatterns</book>
    <book>JS4PHP</book>
  </books>
</response>
```

First, you can see how JSON is lighter in terms of the number of bytes. But, the main benefit is not the smaller byte size but the fact that it's trivial to work with JSON in JavaScript. Let's say you've made an XHR request and have received a JSON string in the `responseText` property of the XHR object. You can convert this string of data into a working JavaScript object by simply using `eval()`:

```
// warning: counter-example
var response = eval('(' + xhr.responseText + ')');
```

Now, you can access the data in obj as object properties:

```
console.log(response.name); // "Stoyan"
console.log(response.books[2]); // "JS4PHP"
```

The problem is that `eval()` is insecure, so it's best if you use the JSON object to parse the JSON data (a fallback for older browsers is available from `http://json.org/`). Creating an object from a JSON string is still trivial:

```
var response = JSON.parse(xhr.responseText);
```

To do the opposite, that is, to convert an object to a JSON string, you use the method `stringify()`:

```
var str = JSON.stringify({hello: "you"});
```

Due to its simplicity, JSON has quickly become popular as a language-independent format for exchanging data and you can easily produce JSON on the server side using your preferred language. In PHP, for example, there are the functions `json_encode()` and `json_decode()` that let you serialize a PHP array or object into a JSON string and vice versa.

Design patterns

The second part of this chapter presents a JavaScript approach to a subset of the design patterns introduced by the book called *Design Patterns: Elements of Reusable Object-Oriented Software*, an influential book most commonly referred to as the *Book of Four* or the *Gang of Four*, or *GoF* (after its four authors). The patterns discussed in the *GoF* book are divided into three groups:

- Creational patterns that deal with how objects are created (instantiated)
- Structural patterns that describe how different objects can be composed in order to provide new functionality
- Behavioral patterns that describe ways for objects to communicate with each other

There are 23 patterns in the *Book of Four* and more patterns have been identified since the book's publication. It's way beyond the scope of this book to discuss all of them, so the remainder of the chapter demonstrates only four, along with examples of their implementation in JavaScript. Remember that the patterns are more about interfaces and relationships rather than implementation. Once you have an understanding of a design pattern, it's often not difficult to implement it, especially in a dynamic language such as JavaScript.

The patterns discussed through the rest of the chapter are:

- Singleton
- Factory
- Decorator
- Observer

Singleton

Singleton is a creational design pattern meaning that its focus is on creating objects. It helps when you want to make sure there is only one object of a given kind (or class). In a classical language this would mean that an instance of a class is only created once and any subsequent attempts to create new objects of the same class would return the original instance.

In JavaScript, because there are no classes, a singleton is the default and most natural pattern. Every object is a singleton object.

The most basic implementation of the singleton in JavaScript is the object literal:

```
var single = {};
```

That was easy, right?

Singleton 2

If you want to use class-like syntax and still implement the singleton pattern, things become a bit more interesting. Let's say you have a constructor called `Logger()` and you want to be able to do something like the following:

```
var my_log = new Logger();
my_log.log('some event');

// ... 1000 lines of code later in a different scope ...

var other_log = new Logger();
other_log.log('some new event');
console.log(other_log === my_log); // true
```

The idea is that although you use `new`, only one instance needs to be created, and this instance is then returned in consecutive calls.

Global variable

One approach would be to use a global variable to store the single instance. Your constructor could look like this:

```
function Logger() {
  if (typeof global_log === "undefined") {
    global_log = this;
  }
  return global_log;
}
```

Using this constructor gives the expected result:

```
var a = new Logger();
var b = new Logger();
console.log(a === b); // true
```

The drawback is, obviously, the use of a global variable. It can be overwritten at any time, even accidentally, and you lose the instance. The opposite, your global variable overwriting someone else's is also possible.

Property of the Constructor

As you know, functions are objects and they have properties. You can assign the single instance to a property of the constructor function as follows:

```
function Logger() {
  if (!Logger.single_instance) {
    Logger.single_instance = this;
  }
  return Logger.single_instance;
}
```

If you write `var a = new Logger()`, `a` points to the newly created `Logger.single_instance` property. A subsequent call `var b = new Logger()` results in `b` pointing to the same `Logger.single_instance` property, which is exactly what you want.

This approach certainly solves the global namespace issue because no global variables are created. The only drawback is that the property of the `Logger` constructor is publicly visible, so it can be overwritten at any time. In such cases, the single instance can be lost or modified. Of course, you can only provide so much protection against fellow programmers shooting themselves in the foot. After all, if someone can mess with the single instance property, they can mess up the `Logger` constructor directly, as well.

In a private property

The solution to the problem of overwriting the publicly visible property is not to use a public property but a private one. You already know how to protect variables with a closure, so as an exercise you can implement this approach to the singleton pattern.

Factory

The factory is another creational design pattern as it deals with creating objects. The factory can help when you have similar types of objects and you don't know in advance which one you want to use. Based on user input or other criteria, your code determines the type of object it needs on the fly.

Let's say you have three different constructors, which implement similar functionality. All the objects they create take a URL but do different things with it. One creates a text DOM node, the second creates a link, and the third an image as follows:

```
var MYAPP = {};
MYAPP.dom = {};
MYAPP.dom.Text = function (url) {
  this.url = url;
  this.insert = function (where) {
    var txt = document.createTextNode(this.url);
    where.appendChild(txt);
  };
};
MYAPP.dom.Link = function (url) {
  this.url = url;
  this.insert = function (where) {
    var link = document.createElement('a');
    link.href = this.url;
    link.appendChild(document.createTextNode(this.url));
    where.appendChild(link);
  };
};
MYAPP.dom.Image = function (url) {
  this.url = url;
  this.insert = function (where) {
    var im = document.createElement('img');
    im.src = this.url;
    where.appendChild(im);
  };
};
```

Using the three different constructors is exactly the same, pass the `url` and call the `insert()` method:

```
var url = 'http://www.phpied.com/images/covers/oojs.jpg';

var o = new MYAPP.dom.Image(url);
o.insert(document.body);

var o = new MYAPP.dom.Text(url);
o.insert(document.body);

var o = new MYAPP.dom.Link(url);
o.insert(document.body);
```

Imagine your program doesn't know in advance which type of object is required. The user decides, during runtime, by clicking on a button for example. If `type` contains the required type of object, you'll need to use an `if` or a `switch`, and do something like this:

```
var o;
if (type === 'Image') {
  o = new MYAPP.dom.Image(url);
}
if (type === 'Link') {
  o = new MYAPP.dom.Link(url);
}
if (type === 'Text') {
  o = new MYAPP.dom.Text(url);
}
o.url = 'http://...';
o.insert();
```

This works fine, but if you have a lot of constructors, the code becomes too lengthy and hard to maintain. Also, if you are creating a library or a framework that allows extensions or plugins, you don't even know the exact names of all the constructor functions in advance. In such cases, it's convenient to have a factory function that takes care of creating an object of the dynamically determined type:

Let's add a factory method to the `MYAPP.dom` utility:

```
MYAPP.dom.factory = function (type, url) {
  return new MYAPP.dom[type](url);
};
```

Now, you can replace the three `if` functions with the simpler code as follows:

```
var image = MYAPP.dom.factory("Image", url);
image.insert(document.body);
```

The example `factory()` method in the previous code was simple, but in a real life scenario you'd want to do some validation against the `type` value (for example, check if `MYAPP.dom[type]` exists) and optionally do some setup work common to all object types (for example, setup the URL all constructors use).

Decorator

The Decorator design pattern is a structural pattern; it doesn't have much to do with how objects are created but rather how their functionality is extended. Instead of using inheritance where you extend in a linear way (parent-child-grandchild), you can have one base object and a pool of different decorator objects that provide extra functionality. Your program can pick and choose which decorators it wants and in which order. For a different program or code path, you might have a different set of requirements and pick different decorators out of the same pool. Take a look at how the usage part of the decorator pattern could be implemented:

```
var obj = {
  doSomething: function () {
    console.log('sure, asap');
  }
  //  ...
};
obj = obj.getDecorator('deco1');
obj = obj.getDecorator('deco13');
obj = obj.getDecorator('deco5');
obj.doSomething();
```

You can see how you can start with a simple object that has a `doSomething()` method. Then you can pick one of the decorator objects you have lying around and can be identified by name. All decorators provide a `doSomething()` method which first calls the same method of the previous decorator and then proceeds with its own code. Every time you add a decorator, you overwrite the base `obj` with an improved version of it. At the end, when you are finished adding decorators, you call `doSomething()`. As a result all of the `doSomething()` methods of all the decorators are executed in sequence. Let's see an example.

Decorating a Christmas tree

Let's illustrate the decorator pattern with an example of decorating a Christmas tree. You start with the `decorate()` method as follows:

```
var tree = {};
tree.decorate = function () {
  alert('Make sure the tree won\'t fall');
};
```

Now, let's implement a `getDecorator()` method which adds extra decorators. The decorators will be implemented as constructor functions, and they'll all inherit from the base `tree` object:

```
tree.getDecorator = function (deco) {
  tree[deco].prototype = this;
  return new tree[deco];
};
```

Now, let's create the first decorator, `RedBalls()`, as a property of `tree` (in order to keep the global namespace cleaner). The red ball objects also provide a `decorate()` method, but they make sure they call their parent's `decorate()` first:

```
tree.RedBalls = function () {
  this.decorate = function () {
    this.RedBalls.prototype.decorate();
    alert('Put on some red balls');
  };
};
```

Similarly, implementing `BlueBalls()` and `Angel()` decorators:

```
tree.BlueBalls = function () {
  this.decorate = function () {
    this.BlueBalls.prototype.decorate();
    alert('Add blue balls');
  };
};
tree.Angel = function () {
  this.decorate = function () {
    this.Angel.prototype.decorate();
    alert('An angel on the top');
  };
};
```

Now, let's add all of the decorators to the base object:

```
tree = tree.getDecorator('BlueBalls');
tree = tree.getDecorator('Angel');
tree = tree.getDecorator('RedBalls');
```

Finally, running the `decorate()` method:

```
tree.decorate();
```

This single call results in the following alerts (in this order):

- Make sure the tree won't fall
- Add blue balls
- An angel on the top
- Add some red balls

As you see, this functionality allows you to have as many decorators as you like, and to choose and combine them in any way you like.

Observer

The observer pattern (also known as the subscriber-publisher pattern) is a behavioral pattern, which means that it deals with how different objects interact and communicate with each other. When implementing the observer pattern you have the following objects:

- One or more publisher objects that announce when they do something important.
- One or more subscribers tuned in to one or more publishers. They listen to what the publishers announce and then act appropriately.

The observer pattern may look familiar to you. It sounds similar to the browser events discussed in the previous chapter, and rightly so, because the browser events are one example application of this pattern. The browser is the publisher, it announces the fact that an event (such as a click) has happened. Your event listener functions that are subscribed to (listen to) this type of event will be notified when the event happens. The browser-publisher sends an event object to all of the subscribers. In your custom implementations you can send any type of data you find appropriate.

There are two subtypes of the observer pattern, push and pull. Push is where the publishers are responsible for notifying each subscriber, and pull is where the subscribers monitor for changes in a publisher's state.

Let's take a look at an example implementation of the push model. Let's keep the observer-related code into a separate object and then use this object as a mix-in, adding its functionality to any other object that decides to be a publisher. In this way any object can become a publisher and any function can become a subscriber. The observer object will have the following properties and methods:

- An array of `subscribers` that are just callback functions
- `addSubscriber()` and `removeSubscriber()` methods that add to, and remove from, the subscribers collection
- A `publish()` method that takes data and calls all subscribers, passing the data to them
- A `make()` method that takes any object and turns it into a publisher by adding all of the methods mentioned previously to it

Here's the observer mix-in object, which contains all the subscription-related methods and can be used to turn any object into a publisher:

```
var observer = {
  addSubscriber: function (callback) {
    if (typeof callback === "function") {
      this.subscribers[this.subscribers.length] = callback;
    }
  },
  removeSubscriber: function (callback) {
    for (var i = 0; i < this.subscribers.length; i++) {
      if (this.subscribers[i] === callback) {
        delete this.subscribers[i];
      }
    }
  },
  publish: function (what) {
    for (var i = 0; i < this.subscribers.length; i++) {
      if (typeof this.subscribers[i] === 'function') {
        this.subscribers[i](what);
      }
    }
  },
  make: function (o) { // turns an object into a publisher
    for (var i in this) {
      if (this.hasOwnProperty(i)) {
        o[i] = this[i];
        o.subscribers = [];
      }
    }
  }
};
```

Now, let's create some publishers. A publisher can be any object and its only duty is to call the `publish()` method whenever something important occurs. Here's a `blogger` object which calls `publish()` every time a new blog posting is ready:

```
var blogger = {
  writeBlogPost: function() {
    var content = 'Today is ' + new Date();
    this.publish(content);
  }
};
```

Another object could be the LA Times newspaper which calls `publish()` when a new newspaper issue is out:

```
var la_times = {
  newIssue: function() {
    var paper = 'Martians have landed on Earth!';
    this.publish(paper);
  }
};
```

Turning these objects into publishers:

```
observer.make(blogger);
observer.make(la_times);
```

Now, let's have two simple objects `jack` and `jill`:

```
var jack = {
  read: function(what) {
    console.log("I just read that " + what)
  }
};
var jill = {
  gossip: function(what) {
    console.log("You didn't hear it from me, but " + what)
  }
};
```

`jack` and `jill` can subscribe to the `blogger` object by providing the callback methods they want to be called when something is published:

```
blogger.addSubscriber(jack.read);
blogger.addSubscriber(jill.gossip);
```

What happens now when the `blogger` writes a new post? The result is that `jack` and `jill` get notified:

```
> blogger.writeBlogPost();
    I just read that Today is Fri Jan 04 2013 19:02:12 GMT-0800 (PST)
    You didn't hear it from me, but Today is Fri Jan 04 2013 19:02:12 GMT-0800 (PST)
```

At any time, `jill` may decide to cancel her subscription. Then, when writing another blog post, the unsubscribed object is no longer notified:

```
> blogger.removeSubscriber(jill.gossip);
> blogger.writeBlogPost();
    I just read that Today is Fri Jan 04 2013 19:03:29 GMT-0800 (PST)
```

`jill` may decide to subscribe to LA Times as an object can be a subscriber to many publishers:

```
> la_times.addSubscriber(jill.gossip);
```

Then, when LA Times publishes a new issue, `jill` gets notified and `jill.gossip()` is executed:

```
> la_times.newIssue();
    You didn't hear it from me, but Martians have landed on Earth!
```

Summary

In this chapter, you learned about common JavaScript coding patterns and learned how to make your programs cleaner, faster, and better at working with other programs and libraries. Then you saw a discussion and sample implementations of a handful of the design patterns from the *Book of Four*. You can see how JavaScript is a fully featured dynamic programming language and that implementing classical patterns in a dynamic loosely typed language is pretty easy. The patterns are, in general, a large topic and you can join the author of this book in a further discussion of the JavaScript patterns at the website `JSPatterns.com` or take a look at the book appropriately named "JavaScript Patterns".

9
Reserved Words

This appendix provides two lists of reserved keywords as defined in ECMAScript 5 (ES5). The first one is the current list of words, and the second is the list of words reserved for future implementations.

There's also a list of words that are no longer reserved, although they used to be in ES3.

You cannot use reserved words as variable names:

```
var break = 1; // syntax error
```

If you use these words as object properties, you have to quote them:

```
var o = {break: 1};   // OK in many browsers, error in IE
var o = {"break": 1}; // Always OK
alert(o.break);       // error in IE
alert(o["break"]);    // OK
```

Keywords

The list of words currently reserved in ES5 is as follows:

`break`

`case`

`catch`

`continue`

`debugger`

`default`

`delete`

`do`

`else`

`finally`

`for`

`function`

`if`

`in`

`instanceof`

`new`

`return`

`switch`

`this`

`throw`

`try`

`typeof`

`var`

`void`

`while`

`with`

Future reserved words

These keywords are not currently used, but are reserved for future versions of the language.

- `class`
- `const`
- `enum`
- `export`
- `extends`
- `implements`

- `import`
- `interface`
- `let`
- `package`
- `private`
- `protected`
- `public`
- `static`
- `super`
- `yield`

Previously reserved words

These words are no longer reserved starting with ES5, but best to stay away for the sake of older browsers.

- `abstract`
- `boolean`
- `byte`
- `char`
- `double`
- `final`
- `float`
- `goto`
- `int`
- `long`
- `native`
- `short`
- `synchronized`
- `throws`
- `transient`
- `volatile`

10
Built-in Functions

This appendix contains a list of the built-in functions (methods of the global object), discussed in *Chapter 3, Functions*.

Function	**Description**

`parseInt()`	Takes two parameters: an input object and radix; then tries to return an integer representation of the input. Doesn't handle exponents in the input. The default radix is `10` (a decimal number). Returns `NaN` on failure. Omitting the radix may lead to unexpected results (for example for inputs such as `08`), so it's best to always specify it.
	`> parseInt('10e+3');`
	10
	`> parseInt('FF');`
	NaN
	`> parseInt('FF', 16);`
	255

`parseFloat()`	Takes a parameter and tries to return a floating-point number representation of it. Understands exponents in the input.
	`> parseFloat('10e+3');`
	10000
	`> parseFloat('123.456test');`
	123.456

Function	Description

`isNaN()`	Abbreviated from "Is Not a Number". Accepts a parameter and returns `true` if the parameter is not a valid number, `false` otherwise. Attempts to convert the input to a number first.
	`> isNaN(NaN);`
	true
	`> isNaN(123);`
	false
	`> isNaN(parseInt('FF'));`
	true
	`> isNaN(parseInt('FF', 16));`
	false

`isFinite()`	Returns `true` if the input is a number (or can be converted to a number), but is not the number `Infinity` or `-Infinity`. Returns `false` for infinity or non-numeric values.
	`> isFinite(1e+1000);`
	false
	`> isFinite(-Infinity);`
	false
	`> isFinite("123");`
	true

`encodeURIComponent()`	Converts the input into a URL-encoded string. For more details on how URL encoding works, refer to the Wikipedia article at `http://en.wikipedia.org/wiki/Url_encode`.
	`> encodeURIComponent ('http://phpied.com/');`
	"http%3A%2F%2Fphpied.com%2F"
	`> encodeURIComponent ('some script?key=v@lue');`
	"some%20script%3Fkey%3Dv%40lue"

`decodeURIComponent()`	Takes a URL-encoded string and decodes it. `> decodeURIComponent('%20%40%20');` **" @ "**

Function	**Description**

`encodeURI()`	URL-encodes the input, but assumes a full URL is given, so returns a valid URL by not encoding the protocol (for example, `http://`) and hostname (for example, `www.phpied.com`).
	`> encodeURI('http://phpied.com/');`
	"http://phpied.com/"
	`> encodeURI('some script?key=v@lue');`
	"some%20script?key=v@lue"

`decodeURI()`	Opposite of `encodeURI()`. `> decodeURI("some%20script?key=v@lue");` **"some script?key=v@lue"**

`eval()`	Accepts a string of JavaScript code and executes it. Returns the result of the last expression in the input string.
	To be avoided where possible.
	`> eval('1 + 2');`
	3
	`> eval('parseInt("123")');`
	123
	`> eval('new Array(1, 2, 3)');`
	[1, 2, 3]
	`> eval('new Array(1, 2, 3); 1 + 2;');`
	3

11 Built-in Objects

This Appendix lists the built-in constructor functions outlined in the **ECMAScript** (**ES**) standard, together with the properties and methods of the objects created by these constructors. ES5-specific APIs are listed separately.

Object

`Object()` is a constructor that creates objects, for example:

```
> var o = new Object();
```

This is the same as using the object literal:

```
> var o = {}; // recommended
```

You can pass anything to the constructor and it will try to guess what it is and use a more appropriate constructor. For example, passing a string to `new Object()` will be the same as using the `new String()` constructor. This is not a recommended practice (it's better to be explicit than let guesses creep in), but still possible.

```
> var o = new Object('something');
> o.constructor;
function String() { [native code] }
```

```
> var o = new Object(123);
> o.constructor;
function Number() { [native code] }
```

All other objects, built-in or custom, inherit from `Object`. So, the properties and methods listed in the following sections apply to all types of objects.

Members of the Object constructor

Have a look at the following table:

Property/method	Description

`Object.prototype`	The prototype of all objects (also an object itself). Anything you add to this prototype will be inherited by all other objects, so be careful.
	`> var s = new String('noodles');`
	`> Object.prototype.custom = 1;`
	1
	`> s.custom;`
	1

The Object.prototype members

Have a look at the following table:

Property/method	Description

`constructor`	Points back to the constructor function used to create the object, in this case, `Object`.
	`> Object.prototype.constructor === Object;`
	true
	`> var o = new Object();`
	`> o.constructor === Object;`
	true

`toString(radix)`	Returns a string representation of the object. If the object happens to be a `Number` object, the radix parameter defines the base of the returned number. The default radix is `10`.
	`> var o = {prop: 1};`
	`> o.toString();`
	"[object Object]"
	`> var n = new Number(255);`
	`> n.toString();`
	"255"
	`> n.toString(16);`
	"ff"

Property/method	Description
`toLocaleString()`	The same as `toString()`, but matching the current locale. Meant to be customized by objects, such as `Date()`, `Number()`, and `Array()` and provide locale-specific values, such as different date formatting. In the case of `Object()` instances as with most other cases, it just calls `toString()`. In browsers, you can figure out the language using the property `language` (or `userLanguage` in IE) of the navigator BOM object: `> navigator.language;` **"en-US"**

`valueOf()`	Returns a primitive representation of `this`, if applicable. For example, `Number` objects return a primitive number and `Date` objects return a timestamp. If no suitable primitive makes sense, it simply returns `this`.
	
	`> var o = {};`
	`> typeof o.valueOf();`
	"object"
	`> o.valueOf() === o;`
	true
	`> var n = new Number(101);`
	`> typeof n.valueOf();`
	"number"
	`> n.valueOf() === n;`
	false
	`> var d = new Date();`
	`> typeof d.valueOf();`
	"number"
	`> d.valueOf();`
	1357840170137

Property/method	**Description**

`hasOwnProperty(prop)`	Returns `true` if a property is an own property of the object, or `false` if it was inherited from the prototype chain. Also returns `false` if the property doesn't exist.
	`> var o = {prop: 1};`
	`> o.hasOwnProperty('prop');`
	true
	`> o.hasOwnProperty('toString');`
	false
	`> o.hasOwnProperty('fromString');`
	false

`isPrototypeOf(obj)`	Returns `true` if an object is used as a prototype of another object. Any object from the prototype chain can be tested, not only the direct creator.
	`> var s = new String('');`
	`> Object.prototype.isPrototypeOf(s);`
	true
	`> String.prototype.isPrototypeOf(s);`
	true
	`> Array.prototype.isPrototypeOf(s);`
	false

`propertyIsEnumerable(prop)`	Returns `true` if a property shows up in a `for-in` loop.
	`> var a = [1, 2, 3];`
	`> a.propertyIsEnumerable('length');`
	false
	`> a.propertyIsEnumerable(0);`
	true

ECMAScript 5 additions to Object

In ECMAScript 3 all object properties can be changed, added, or deleted at any time, except for a few built-in properties (for example, `Math.PI`). In ES5 you have the ability to define properties that cannot be changed or deleted—a privilege previously reserved for built-ins. ES5 introduces the concept of **property descriptors** that give you tighter control over the properties you define.

Think of a property descriptor as an object that specifies the features of a property. The syntax to describe these features is a regular object literal, so property descriptors have properties and methods of their own, but let's call them **attributes** to avoid confusion. The attributes are:

- `value` – what you get when you access the property
- `writable` – can you change this property
- `enumerable` – should it appear in `for-in` loops
- `configurable` – can you delete it
- `set()` – a function called any time you update the value
- `get()` – called when you access the value of the property

Further, there's a distinction between **data descriptors** (you define the properties `enumerable`, `configurable`, `value`, and `writable`) and **accessor descriptors** (you define `enumerable`, `configurable`, `set()`, and `get()`). If you define `set()` or `get()`, the descriptor is considered an accessor and attempting to define value or writable will raise an error.

Defining a regular old school ES3-style property:

```
var person = {};
person.legs = 2;
```

The same using an ES5 data descriptor:

```
var person = {};
Object.defineProperty(person, "legs", {
  value: 2,
  writable: true,
  configurable: true,
  enumerable: true
});
```

The value of `value` if set to `undefined` by default, all others are `false`. So, you need to set them to `true` explicitly if you want to be able to change this property later.

Or, the same property using an ES5 accessor descriptor:

```
var person = {};
Object.defineProperty(person, "legs", {
  set: function (v) {this.value = v;},
  get: function (v) {return this.value;},
  configurable: true,
  enumerable: true
});
person.legs = 2;
```

As you can see property descriptors are a lot more code, so you only use them if you really want to prevent someone from mangling your property, and also you forget about backwards compatibility with ES3 browsers because, unlike additions to `Array.prototype` for example, you cannot "shim" this feature in old browsers.

And the power of the descriptors in action (defining a nonmalleable property):

```
> var person = {};
> Object.defineProperty(person, 'heads', {value: 1});
> person.heads = 0;
0

> person.heads;
1

> delete person.heads;
false

> person.heads;
1
```

The following is a list of all ES5 additions to `Object`:

Property/method	Description

`Object.getPrototypeOf(obj)`	While in ES3 you have to guess what is the prototype of a given object using the method `Object.prototype.isPrototypeOf()`, in ES5 you can directly ask "Who is your prototype?"
	`> Object.getPrototypeOf([]) ===`
	`    Array.prototype;`
	true

Property/method	**Description**

`Object.create(obj, descr)`	Discussed in *Chapter 6*, *Inheritance*. Creates a new object, sets its prototype and defines properties of that object using property descriptors (discussed earlier).
	`> var parent = {hi: 'Hello'};`
	`> var o = Object.create(parent, {`
	`    prop: {`
	`      value: 1`
	`    }`
	`  });`
	`> o.hi;`
	"Hello"
	It even lets you create a completely blank object, something you cannot do in ES3.
	`> var o = Object.create(null);`
	`> typeof o.toString;`
	"undefined"

`Object.getOwnPropertyDescriptor(obj, property)`	Allows you to inspect how a property was defined. You can even peek into the built-ins and see all these previously hidden attributes.
	`> Object.getOwnPropertyDescriptor(`
	`    Object.prototype, 'toString');`
	Object
	configurable: true
	enumerable: false
	value: function toString() { [native code] }
	writable: true

`Object.getOwnPropertyNames(obj)`	Returns an array of all own property names (as strings), enumerable or not. Use `Object.keys()` to get only enumerable ones.
	`> Object.getOwnPropertyNames(`
	`    Object.prototype);`
	["constructor", "toString", "toLocaleString", "valueOf",...

`Object.defineProperty(obj, descriptor)`	Defines a property of an object using a property descriptor. See the discussion preceding this table.

Property/method	**Description**

`Object.defineProperties(obj, descriptors)`	The same as `defineProperty()`, but lets you define multiple properties at once.
	`> var glass =`
	`  Object.defineProperties({}, {`
	`    "color": {`
	`      value: "transparent",`
	`      writable: true`
	`    },`
	`    "fullness": {`
	`      value: "half",`
	`      writable: false`
	`    }`
	`  });`
	
	`> glass.fullness;`
	"half"

`Object.preventExtensions(obj)`
 	
`Object.isExtensible(obj)`
	`preventExtensions()` disallows adding further properties to an object and `isExtensible()` checks whether you can add properties.
	`> var deadline = {};`
	`> Object.isExtensible(deadline);`
	true
	`> deadline.date = "yesterday";`
	"yesterday"
	`> Object.`
	`preventExtensions(deadline);`
	`> Object.isExtensible(deadline);`
	false
	`> deadline.date = "today";`
	"today"
	`> deadline.date;`
	"today"
	Attempting to add properties to a non-extensible object is not an error, but simply doesn't work:
	`> deadline.report = true;`
	`> deadline.report;`
	undefined

Property/method	Description

`Object.seal(obj)`
 	
`Object.isSealed(obj)`
	`seal()` does the same as `preventExtensions ()` and additionally makes all existing properties non-configurable.
	
	This means you *can* change the value of an existing property, but you *cannot* delete it or reconfigure it (using `defineProperty()` won't work). So you cannot, for example, make an enumerable property non-enumerable.

`Object.freeze(obj)`
 	
`Object.isFrozen(obj)`
	Everything that `seal()` does plus prevents changing the values of properties.
	`> var deadline = Object.freeze(`
	`    {date: "yesterday"});`
	`> deadline.date = "tomorrow";`
	`> deadline.excuse = "lame";`
	`> deadline.date;`
	"yesterday"
	`> deadline.excuse;`
	undefined
	`> Object.isSealed(deadline);`
	true

`Object.keys(obj)`	An alternative to a `for-in` loop. Returns only own properties (unlike `for-in`). The properties need to be enumerable in order to show up (unlike `Object.getOwnPropertyNames()`). The return value is an array of strings.
	`> Object.prototype.customProto =`
	`101;`
	`> Object.getOwnPropertyNames(`
	`    Object.prototype);`
	["constructor", "toString", ..., "customProto"]
	`> Object.keys(Object.prototype);`
	["customProto"]
	`> var o = {own: 202};`
	`> o.customProto;`
	`101`
	`> Object.keys(o);`
	["own"]

Array

The `Array` constructor creates array objects:

```
> var a = new Array(1, 2, 3);
```

This is the same as the array literal:

```
> var a = [1, 2, 3]; //recommended
```

When you pass only one numeric value to the `Array` constructor, it's assumed to be the array length.

```
> var un = new Array(3);
> un.length;
3
```

You get an array with the desired length and if you ask for the value of each of the array elements, you get `undefined`.

```
> un;
[undefined, undefined, undefined]
```

There is a subtle difference between an array full of elements and an array with no elements, but just length:

```
> '0' in a;
true

> '0' in un;
false
```

This difference in the `Array()` constructor's behavior when you specify one versus more parameters can lead to unexpected behavior. For example, the following use of the array literal is valid:

```
> var a = [3.14];
> a;
[3.14]
```

However, passing the floating-point number to the `Array` constructor is an error:

```
> var a = new Array(3.14);
Range Error: invalid array length
```

The Array.prototype members

Property/method	**Description**
`length`	The number of elements in the array. `> [1, 2, 3, 4].length;` **4**
`concat(i1, i2, i3,...)`	Merges arrays together. `> [1, 2].concat([3, 5], [7, 11]);` **[1, 2, 3, 5, 7, 11]**

`join(separator)`	Turns an array into a string. The separator parameter is a string with comma as the default value.
	`> [1, 2, 3].join();`
	"1,2,3"
	`> [1, 2, 3].join('|');`
	"1|2|3"
	`> [1, 2, 3].join(' is less than ');`
	"1 is less than 2 is less than 3"

`pop()`	Removes the last element of the array and returns it.
	`> var a = ['une', 'deux', 'trois'];`
	`> a.pop();`
	"trois"
	`> a;`
	["une", "deux"]

`push(i1, i2, i3,...)`	Appends elements to the end of the array and returns the length of the modified array.
	`> var a = [];`
	`> a.push('zig', 'zag', 'zebra','zoo');`
	4

`reverse()`	Reverses the elements of the array and returns the modified array.
	`> var a = [1, 2, 3];`
	`> a.reverse();`
	[3, 2, 1]
	`> a;`
	[3, 2, 1]

Property/method	**Description**

`shift()`	Like `pop()` but removes the first element, not the last.
	`> var a = [1, 2, 3];`
	`> a.shift();`
	1
	`> a;`
	[2, 3]

`slice(start_index, end_index)`	Extracts a piece of the array and returns it as a new array, without modifying the source array.
	`> var a = ['apple', 'banana',`
	`              'js', 'css', 'orange'];`
	`> a.slice(2,4);`
	["js", "css"]
	`> a;`
	["apple", "banana", "js", "css", "orange"]

`sort(callback)`	Sorts an array. Optionally accepts a callback function for custom sorting. The callback function receives two array elements as arguments and should return `0` if they are equal, a positive number if the first is greater and a negative number if the second is greater.
	An example of a custom sorting function that does a proper *numeric* sort (since the default is *character* sorting):
	`function customSort(a, b) {`
	`  if (a > b) return 1;`
	`  if (a < b) return -1;`
	`  return 0;`
	`}`
	`Example use of sort():`
	`> var a = [101, 99, 1, 5];`
	`> a.sort();`
	[1, 101, 5, 99]
	`> a.sort(customSort);`
	[1, 5, 99, 101]
	`> [7, 6, 5, 9].sort(customSort);`
	[5, 6, 7, 9]

Property/method	Description

`splice(start, delete_count, i1, i2, i3,...)`	Removes and adds elements at the same time. The first parameter is where to start removing, the second is how many items to remove and the rest of the parameters are new elements to be inserted in the place of the removed ones.
	`> var a = ['apple', 'banana',`
	`              'js', 'css', 'orange'];`
	`> a.splice(2, 2, 'pear', 'pineapple');`
	["js", "css"]
	`> a;`
	["apple", "banana", "pear", "pineapple", "orange"]

`unshift(i1, i2, i3,...)`	Like `push()` but adds the elements at the beginning of the array as opposed to the end. Returns the length of the modified array.
	`> var a = [1, 2, 3];`
	`> a.unshift('one', 'two');`
	5
	`> a;`
	["one", "two", 1, 2, 3]

ECMAScript 5 additions to Array

Property/method	**Description**

`Array.isArray(obj)`	Tells if an object is an array because `typeof` is not good enough:
	`> var arraylike = {0: 101, length: 1};`
	`> typeof arraylike;`
	"object"
	`> typeof [];`
	"object"
	Neither is duck-typing (if it walks like a duck and quacks like a duck, it must be a duck):
	`typeof arraylike.length;`
	"number"
	In ES3 you need the verbose:
	`> Object.prototype.toString.call([]) === "[object Array]";`
	true
	`> Object.prototype.toString.call (arraylike) === "[object Array]";`
	false
	In ES5 you get the shorter:
	`Array.isArray([]);`
	true
	`Array.isArray(arraylike);`
	false

`Array.prototype. indexOf(needle, idx)`	Searches the array and returns the index of the first match. Returns `-1` if there's no match. Optionally can search starting from a specified index.
	`> var ar = ['one', 'two', 'one', 'two'];`
	`> ar.indexOf('two');`
	1
	`> ar.indexOf('two', 2);`
	3
	`> ar.indexOf('toot');`
	-1

Property/method	**Description**

`Array.prototype.`
`lastIndexOf(needle,`
`idx)`
	Like `indexOf()` only searches from the end.
	`> var ar = ['one', 'two', 'one', 'two'];`
	`> ar.lastIndexOf('two');`
	3
	`> ar.lastIndexOf('two', 2);`
	1
	`> ar.indexOf('toot');`
	-1

`Array.prototype.`
`forEach(callback,`
`this_obj)`
	An alternative to a `for` loop. You specify a callback function that will be called for each element of the array. The callback function gets the arguments: the element, its index and the whole array.
	`> var log = console.log.bind(console);`
	`> var ar = ['itsy', 'bitsy', 'spider'];`
	`> ar.forEach(log);`
	itsy 0 ["itsy", "bitsy", "spider"]
	bitsy 1 ["itsy", "bitsy", "spider"]
	spider 2 ["itsy", "bitsy", "spider"]
	Optionally, you can specify a second parameter: the object to be bound to this inside the callback function. So this works too:
	`> ar.forEach(console.log, console);`

Property/method	**Description**

`Array.prototype.every(callback, this_obj)`	You provide a callback function that tests each element of the array. Your callback is given the same arguments as `forEach()` and it must return `true` or `false` depending on whether the given element satisfies your test.
	
	If all elements satisfy your test, `every()` returns `true`. If at least one doesn't, `every()` returns `false`.
	
	`> function hasEye(el, idx, ar) {`
	`    return el.indexOf('i') !== -1;`
	`  }`
	
	`> ['itsy', 'bitsy', 'spider'].`
	`     every(hasEye);`
	true
	`> ['eency', 'weency', 'spider'].`
	`     every(hasEye);`
	false
	
	If at some point during the loop it becomes clear that the result will be false, the loop stops and returns false.
	
	`> [1,2,3].every(function (e) {`
	`    console.log(e);`
	`    return false;`
	`  });`
	1
	false

`Array.prototype.some(callback, this_obj)`	Like `every()` only it returns true if at least one element satisfies your test:
	
	`> ['itsy', 'bitsy', 'spider'].`
	`     some(hasEye);`
	true
	`> ['eency', 'weency', 'spider'].`
	`     some(hasEye);`
	true

`Array.prototype.filter(callback, this_obj)`	Similar to some() and every() but it returns a new array of all elements that satisfy your test:
	
	`> ['itsy', 'bitsy', 'spider'].`
	`     filter(hasEye);`
	["itsy", "bitsy", "spider"]
	`> ['eency', 'weency', 'spider'].`
	`     filter(hasEye);`
	["spider"]

Property/method	Description

`Array.prototype.map(callback, this_obj)`	Similar to `forEach()` because it executes a callback for each element, but additionally it constructs a new array with the returned values of your callback and returns it. Let's capitalize all strings in an array:
	`> function uc(element, index, array) {`
	`    return element.toUpperCase();`
	`  }`
	`> ['eency', 'weency', 'spider'].map(uc);`
	["EENCY", "WEENCY", "SPIDER"]

`Array.prototype.reduce(callback, start)`	Executes your callback for each element of the array. Your callback returns a value. This value is passed back to your callback with the next iteration. The whole array is eventually reduced to a single value.
	`> function sum(res, element, idx, arr) {`
	`    return res + element;`
	`  }`
	`> [1, 2, 3].reduce(sum);`
	6
	Optionally, you can pass a start value which will be used by the first callback call:
	`> [1, 2, 3].reduce(sum, 100);`
	106

`Array.prototype.reduceRight(callback, start)`	Like `reduce()` but loops from the end of the array.
	`> function concat(result_so_far, el) {`
	`    return "" + result_so_far + el;`
	`  }`
	
	`> [1, 2, 3].reduce(concat);`
	"123"
	`> [1, 2, 3].reduceRight(concat);`
	"321"

Function

JavaScript functions are objects. They can be defined using the `Function` constructor, like so:

```
var sum = new Function('a', 'b', 'return a + b;');
```

This is a (generally not recommended) alternative to the function literal (also known as function expression):

```
var sum = function (a, b) {
  return a + b;
};
```

Or, the more common function definition:

```
function sum(a, b) {
  return a + b;
}
```

The Function.prototype members

Property/Method	Description

`apply(this_obj, params_array)`	Allows you to call another function while overwriting the other function's this value. The first parameter that `apply()` accepts is the object to be bound to this inside the function and the second is an array of arguments to be send to the function being called.
	`function whatIsIt(){`
	`  return this.toString();`
	`}`
	`> var myObj = {};`
	`> whatIsIt.apply(myObj);`
	"[object Object]"
	`> whatIsIt.apply(window);`
	"[object Window]"

`call(this_obj, p1, p2, p3, ...)`	The same as `apply()` but accepts arguments one by one, as opposed to as one array.

`length`	The number of parameters the function expects.
	`> parseInt.length;`
	2
	If you forget the difference between call() and apply():
	`> Function.prototype.call.length;`
	1
	`> Function.prototype.apply.length;`
	2
	The call() property's length is 1 because all arguments except the first one are optional.

ECMAScript 5 additions to a function

Property/method	Description
`Function.prototype.bind()`	When you want to call a function that uses this internally and you want to define what this is. The methods `call()` and `apply()` invoke the function while `bind()` returns a new function. Useful when you provide a method as a callback to a method of another object and and you want this to be an object of your choice. `> whatIsIt.apply(window);` **"[object Window]"**

Boolean

The `Boolean` constructor creates Boolean objects (not to be confused with Boolean primitives). The Boolean objects are not that useful and are listed here for the sake of completeness.

```
> var b = new Boolean();
> b.valueOf();
false

> b.toString();
"false"
```

A Boolean object is not the same as a Boolean primitive value. As you know, all objects are truthy:

```
> b === false;
false

> typeof b;
"object"
```

Boolean objects don't have any properties other than the ones inherited from `Object`.

Number

This creates number objects:

```
> var n = new Number(101);
> typeof n;
"object"

> n.valueOf();
101
```

The `Number` objects are not primitive objects, but if you use any `Number.prototype` method on a primitive number, the primitive will be converted to a `Number` object behind the scenes and the code will work.

```
> var n = 123;
> typeof n;
"number"

> n.toString();
"123"
```

Used without `new`, the `Number` constructor returns a primitive number.

```
> Number("101");
101

> typeof Number("101");
"number"

> typeof new Number("101");
"object"
```

Members of the Number constructor

Property/method	Description
`Number.MAX_VALUE`	A constant property (cannot be changed) that contains the maximum allowed number. `> Number.MAX_VALUE;` **1.7976931348623157e+308**
`Number.MIN_VALUE`	The smallest number you can work with in JavaScript. `> Number.MIN_VALUE;` **5e-324**

`Number.NaN`	Contains the Not A Number number. The same as the global NaN.
	`> Number.NaN;`
	NaN
	NaN is not equal to anything including itself.
	`> Number.NaN === Number.NaN;`
	false

`Number.POSITIVE_INFINITY`	The same as the global `Infinity` number.
`Number.NEGATIVE_INFINITY`	The same as `-Infinity`.

The Number.prototype members

Property/method	Description

`toFixed(fractionDigits)`	Returns a string with the fixed-point representation of the number. Rounds the returned value.
	`> var n = new Number(Math.PI);`
	`> n.valueOf();`
	3.141592653589793
	`> n.toFixed(3);`
	"3.142"

`toExponential (fractionDigits)`	Returns a string with exponential notation representation of the number object. Rounds the returned value.
	`> var n = new Number(56789);`
	`> n.toExponential(2);`
	"5.68e+4"

`toPrecision(precision)`	String representation of a number object, either exponential or fixed-point, depending on the number object.
	`> var n = new Number(56789);`
	`> n.toPrecision(2);`
	"5.7e+4"
	`> n.toPrecision(5);`
	"56789"
	`> n.toPrecision(4);`
	"5.679e+4"
	`> var n = new Number(Math.PI);`
	`> n.toPrecision(4);`
	"3.142"

String

The `String()` constructor creates string objects. Primitive strings are turned into objects behind the scenes if you call a method on them as if they were objects. Omitting `new` gives you primitive strings.

Creating a string object and a string primitive:

```
> var s_obj = new String('potatoes');
> var s_prim = 'potatoes';
> typeof s_obj;
"object"
```

```
> typeof s_prim;
"string"
```

The object and the primitive are not equal when compared by type with `===`, but they are when compared with `==` which does type coercion:

```
> s_obj === s_prim;
false
```

```
> s_obj == s_prim;
true
```

`length` is a property of the string objects:

```
> s_obj.length;
8
```

If you access `length` on a primitive string, the primitive is converted to an object behind the scenes and the operation is successful:

```
> s_prim.length;
8
```

String literals work fine too:

```
> "giraffe".length;
7
```

Members of the String constructor

Property/method	**Description**
`String.fromCharCode (code1, code2, code3, ...)`	Returns a string created using the Unicode values of the input: `> String.fromCharCode(115, 99, 114, 105, 112, 116);` **"script"**

The String.prototype members

Property/method	Description
`length`	The number of characters in the string. `> new String('four').length;` **4**

`charAt(position)`	Returns the character at the specified position.
	Positions start at 0.
	`> "script".charAt(0);`
	"s"
	Since ES5, it's also possible to use array notation for the same purpose. (This feature has been long supported in many browsers before ES5, but not IE)
	`> "script"[0];`
	"s"

`charCodeAt(position)`	Returns the numeric code (Unicode) of the character at the specified position. `> "script".charCodeAt(0);` **115**
`concat(str1, str2, ....)`	Return a new string glued from the input pieces. `> "".concat('zig', '-', 'zag');` **"zig-zag"**

`indexOf(needle, start)`	If the needle matches a part of the string, the position of the match is returned. The optional second parameter defines where the search should start from. Returns `-1` if no match is found.
	`> "javascript".indexOf('scr');`
	4
	`> "javascript".indexOf('scr', 5);`
	-1

`lastIndexOf(needle, start)`	Same as `indexOf()` but starts the search from the end of the string. The last occurrence of `a`: `> "javascript".lastIndexOf('a');` **3**

Property/method	Description

`localeCompare (needle)`	Compares two strings in the current locale. Returns `0` if the two strings are equal, `1` if the needle gets sorted before the string object, `-1` otherwise.
	`> "script".localeCompare('crypt');`
	1
	`> "script".localeCompare('sscript');`
	-1
	`> "script".localeCompare('script');`
	0

`match(regexp)`	Accepts a regular expression object and returns an array of matches. `> "R2-D2 and C-3PO".match(/[0-9]/g);` **["2", "2", "3"]**

`replace(needle, replacement)`	Allows you to replace the matching results of a regexp pattern. The replacement can also be a callback function. Capturing groups are available as `$1, $2,...$9`.
	`> "R2-D2".replace(/2/g, '-two');`
	"R-two-D-two"
	`> "R2-D2".replace(/(2)/g, '$1$1');`
	"R22-D22"

`search(regexp)`	Returns the position of the first regular expression match. `> "C-3PO".search(/[0-9]/);` **2**

`slice(start, end)`	Returns the part of a string identified by the start and end positions. If `start` is negative, the start position is `length + start`, similarly if the `end` parameter is negative, the end position is length + end.
	`> "R2-D2 and C-3PO".slice(4, 13);`
	"2 and C-3"
	`> "R2-D2 and C-3PO".slice(4, -1);`
	"2 and C-3P"

`split(separator, limit)`	Turns a string into an array. The second parameter, limit, is optional. As with `replace()`, `search()`, and `match()`, the separator is a regular expression but can also be a string.
	`> "1,2,3,4".split(/,/);`
	["1", "2", "3", "4"]
	`> "1,2,3,4".split(',', 2);`
	["1", "2"]

Property/method	Description

`substring(start, end)`	Similar to `slice()`. When start or end are negative or invalid, they are considered 0. If they are greater than the string length, they are considered to be the length. If `end` is greater than `start`, their values are swapped.
	`> "R2-D2 and C-3PO".substring(4, 13);`
	"2 and C-3"
	`> "R2-D2 and C-3PO".substring(13, 4);`
	"2 and C-3"

`toLowerCase()`
`toLocaleLowerCase()`
	Transforms the string to lowercase.
	`> "Java".toLowerCase();`
	"java"

`toUpperCase()`
`toLocaleUpperCase()`
	Transforms the string to uppercase.
	`> "Script".toUpperCase();`
	"SCRIPT"

ECMAScript 5 additions to String

Property/method	Description

`String.prototype.trim()`	Instead of using a regular expression to remove whitespace before and after a string (as in ES3), you have a `trim()` method in ES5.
	`> " \t beard \n".trim();`
	"beard"
	`Or in ES3:`
	`> " \t beard \n".replace(/\s/g, "");`
	"beard"

Date

The `Date` constructor can be used with several types of input:

You can pass values for year, month, date of the month, hour, minute, second, and millisecond, like so:

```
> new Date(2015, 0, 1, 13, 30, 35, 505);
Thu Jan 01 2015 13:30:35 GMT-0800 (PST)
```

- You can skip any of the input parameters, in which case they are assumed to be 0. Note that month values are from 0 (January) to 11 (December), hours are from 0 to 23, minutes and seconds 0 to 59, and milliseconds 0 to 999.

- You can pass a timestamp:

```
> new Date(1420147835505);
Thu Jan 01 2015 13:30:35 GMT-0800 (PST)
```

- If you don't pass anything, the current date/time is assumed:

```
> new Date();
Fri Jan 11 2013 12:20:45 GMT-0800 (PST)
```

- If you pass a string, it's parsed in an attempt to extract a possible date value:

```
> new Date('May 4, 2015');
Mon May 04 2015 00:00:00 GMT-0700 (PDT)
```

Omitting `new` gives you a string version of the current date:

```
> Date() === new Date().toString();
true
```

Members of the Date constructor

Property/method	Description

`Date.parse(string)`	Similar to passing a string to new `Date()` constructor, this method parses the input string in an attempt to extract a valid date value. Returns a timestamp on success, `NaN` on failure:
	`> Date.parse('May 5, 2015');`
	1430809200000
	`> Date.parse('4th');`
	NaN

`Date.UTC(year, month, date, hours, minutes, seconds, ms)`	Returns a timestamp but in UTC (Coordinated Universal Time), not in local time.
	`> Date.UTC`
	`    (2015, 0, 1, 13, 30, 35, 505);`
	1420119035505

The Date.prototype members

Property/method	Description/example
`toUTCString()`	Same as `toString()` but in universal time. Here's how Pacific Standard (PST) local time differs from UTC: `> var d = new Date(2015, 0, 1);` `> d.toString();` **"Thu Jan 01 2015 00:00:00 GMT-0800 (PST)"** `> d.toUTCString();` **"Thu, 01 Jan 2015 08:00:00 GMT"**
`toDateString()`	Returns only the date portion of `toString()`: `> new Date(2015, 0, 1).toDateString();` **"Thu Jan 01 2010"**
`toTimeString()`	Returns only the time portion of `toString()`: `> new Date(2015, 0, 1).toTimeString();` **"00:00:00 GMT-0800 (PST)"**
`toLocaleString()` `toLocaleDateString()` `toLocaleTimeString()`	Equivalent to `toString()`, `toDateString()`, and `toTimeString()` respectively, but in a friendlier format, according to the current user's locale. `> new Date(2015, 0, 1).toString();` **"Thu Jan 01 2015 00:00:00 GMT-0800 (PST)"** `> new Date(2015, 0, 1).toLocaleString();` **"1/1/2015 12:00:00 AM"**
`getTime()` `setTime(time)`	Get or set the time (using a timestamp) of a date object. The following example creates a date and moves it one day forward: **> var d = new Date(2015, 0, 1);** **> d.getTime();** **1420099200000** **> d.setTime(d.getTime() + 1000 * 60 * 60 * 24);** **1420185600000** **> d.toLocaleString();** **"Fri Jan 02 2015 00:00:00 GMT-0800 (PST)"**

Property/method	**Description/example**
`getFullYear()` `getUTCFullYear()` `setFullYear(year, month, date)` `setUTCFullYear(year, month, date)`	Get or set a full year using local or UTC time. There is also `getYear()` but it is not Y2K compliant, so use `getFullYear()` instead. `> var d = new Date(2015, 0, 1);` `> d.getYear();` **115** `> d.getFullYear();` **2015** `> d.setFullYear(2020);` **1577865600000** `> d;` **Wed Jan 01 2020 00:00:00 GMT-0800 (PST)**
`getMonth()` `getUTCMonth()` `setMonth(month, date)` `setUTCMonth(month, date)`	Get or set the month, starting from 0 (January): `> var d = new Date(2015, 0, 1);` `> d.getMonth();` **0** `> d.setMonth(11);` **1448956800000** `> d.toLocaleDateString();` **"12/1/2015"**
`getDate()` `getUTCDate()` `setDate(date)` `setUTCDate(date)`	Get or set the date of the month. `> var d = new Date(2015, 0, 1);` `> d.toLocaleDateString();` **"1/1/2015"** `> d.getDate();` **1** `> d.setDate(31);` **1422691200000** `> d.toLocaleDateString();` **"1/31/2015"**

Property/method	Description/example

`getHours()`
`getUTCHours()`
`setHours(hour, min, sec, ms)`
`setUTCHours(hour, min, sec, ms)`
`getMinutes()`
`getUTCMinutes()`
`setMinutes(min, sec, ms)`
`setUTCMinutes(min, sec, ms)`
`getSeconds()`
`getUTCSeconds()`
`setSeconds(sec, ms)`
`setUTCSeconds(sec, ms)`
`getMilliseconds()`
`getUTCMilliseconds()`
`setMilliseconds(ms)`
`setUTCMilliseconds(ms)`
	Get or set the hour, minutes, seconds, milliseconds, all starting from 0.
	`> var d = new Date(2015, 0, 1);`
	`> d.getHours() + ':' + d.getMinutes();`
	"0:0"
	`> d.setMinutes(59);`
	1420102740000
	`> d.getHours() + ':' + d.getMinutes();`
	"0:59"

`getTimezoneOffset()`	Returns the difference between local and universal (UTC) time, measured in minutes. For example, the difference between PST (Pacific Standard Time) and UTC:
	`> new Date().getTimezoneOffset();`
	480
	`> 420 / 60; // hours`
	8

Property/method	Description/example

`getDay()`
`getUTCDay()`
	Returns the day of the week, starting from 0 (Sunday):
	`> var d = new Date(2015, 0, 1);`
	`> d.toDateString();`
	"Thu Jan 01 2015"
	`> d.getDay();`
	4
	`> var d = new Date(2015, 0, 4);`
	`> d.toDateString();`
	"Sat Jan 04 2015"
	`> d.getDay();`
	0

ECMAScript 5 additions to Date

Property/method	Description
`Date.now()`	A convenient way to get the current timestamp: `> Date.now() === new Date().getTime();` **true**

`Date.prototype.`
`toISOString()`
	Yet another toString().
	`> var d = new Date(2015, 0, 1);`
	`> d.toString();`
	"Thu Jan 01 2015 00:00:00 GMT-0800 (PST)"
	`> d.toUTCString();`
	"Thu, 01 Jan 2015 08:00:00 GMT"
	`> d.toISOString();`
	"2015-01-01T00:00:00.000Z"

`Date.prototype.`
`toJSON()`
	Used by JSON.stringify() (refer to the end of this appendix) and returns the same as toISOString().
	`> var d = new Date();`
	`> d.toJSON() === d.toISOString();`
	true

Math

`Math` is different from the other built-in objects because it cannot be used as a constructor to create objects. It's just a collection of static functions and constants. Some examples to illustrate the differences are as follows:

```
> typeof Date.prototype;
"object"

> typeof Math.prototype;
"undefined"

> typeof String;
"function"

> typeof Math;
"object"
```

Members of the Math object

Property/method	Description

`Math.E`
`Math.LN10`
`Math.LN2`
`Math.LOG2E`
`Math.LOG10E`
`Math.PI`
`Math.SQRT1_2`
`Math.SQRT2`
	These are some useful math constants, all read-only. Here are their values:
	`> Math.E;`
	2.718281828459045
	`> Math.LN10;`
	2.302585092994046
	`> Math.LN2;`
	0.6931471805599453
	`> Math.LOG2E;`
	1.4426950408889634
	`> Math.LOG10E;`
	0.4342944819032518
	`> Math.PI;`
	3.141592653589793
	`> Math.SQRT1_2;`
	0.7071067811865476
	`> Math.SQRT2;`
	1.4142135623730951

Property/method	**Description**

`Math.acos(x)`
`Math.asin(x)`
`Math.atan(x)`
`Math.atan2(y, x)`
`Math.cos(x)`
`Math.sin(x)`
`Math.tan(x)`
	Trigonometric functions

`Math.round(x)`
`Math.floor(x)`
`Math.ceil(x)`
	`round()` gives you the nearest integer, `ceil()` rounds up, and `floor()` rounds down:
	`> Math.round(5.5);`
	6
	`> Math.floor(5.5);`
	5
	`> Math.ceil(5.1);`
	6

`Math.max(num1, num2, num3, ...)`
`Math.min(num1, num2, num3, ...)`
	`max()` returns the largest and `min()` returns the smallest of the numbers passed to them as arguments. If at least one of the input parameters is `NaN`, the result is also `NaN`.
	`> Math.max(4.5, 101, Math.PI);`
	101
	`> Math.min(4.5, 101, Math.PI);`
	3.141592653589793

`Math.abs(x)`	Absolute value.
	`> Math.abs(-101);`
	101
	`> Math.abs(101);`
	101

`Math.exp(x)`	Exponential function: `Math.E` to the power of `x`. `> Math.exp(1) === Math.E;` **true**
`Math.log(x)`	Natural logarithm of `x`. `> Math.log(10) === Math.LN10;` **true**

Property/method	Description

`Math.sqrt(x)`	Square root of `x`.
	`> Math.sqrt(9);`
	3
	`> Math.sqrt(2) === Math.SQRT2;`
	true

`Math.pow(x, y)`	`x` to the power of `y`. `> Math.pow(3, 2);` **9**

`Math.random()`	Random number between 0 and 1 (including 0).
	`> Math.random();`
	0.8279076443185321
	`For an random integer in a range, say between 10 and 100:`
	`> Math.round(Math.random() * 90 + 10);`
	79

RegExp

You can create a regular expression object using the `RegExp()` constructor. You pass the expression pattern as the first parameter and the pattern modifiers as the second.

```
> var re = new RegExp('[dn]o+dle', 'gmi');
```

This matches "noodle", "doodle", "doooodle", and so on. It's equivalent to using the regular expression literal:

```
> var re = ('/[dn]o+dle/gmi'); // recommended
```

Chapter 4, Objects and *Chapter 12, Regular Expressions* contains more information on regular expressions and patterns.

The RegExp.prototype members

Property/method	Description
`global`	Read-only. `true` if the `g` modifier was set when creating the regexp object.
`ignoreCase`	Read-only. `true` if the `i` modifier was set when creating the regexp object.
`multiline`	Read-only. `true` if the `m` modifier was set when creating the regexp object

Property/method	Description
`lastIndex`	Contains the position in the string where the next match should start. `test()` and `exec()` set this position after a successful match. Only relevant when the `g` (global) modifier was used.
	`> var re = /[dn]o+dle/g;`
	`> re.lastIndex;`
	0
	`> re.exec("noodle doodle");`
	["noodle"]
	`> re.lastIndex;`
	6
	`> re.exec("noodle doodle");`
	["doodle"]
	`> re.lastIndex;`
	13
	`> re.exec("noodle doodle");`
	null
	`> re.lastIndex;`
	0
`source`	Read-only. Returns the regular expression pattern (without the modifiers).
	`> var re = /[nd]o+dle/gmi;`
	`> re.source;`
	"[nd]o+dle"
`exec(string)`	Matches the input string with the regular expression. A successful match returns an array containing the match and any capturing groups. With the `g` modifier, it matches the first occurrence and sets the `lastIndex` property. Returns `null` when there's no match.
	`> var re = /([dn])(o+)dle/g;`
	`> re.exec("noodle doodle");`
	["noodle", "n", "oo"]
	`> re.exec("noodle doodle");`
	["doodle", "d", "oo"]
	The arrays returned by exec() have two additional properties: index (of the match) and input (the input string being searched).
`test(string)`	Same as `exec()` but only returns true or false.
	`> /noo/.test('Noodle');`
	false
	`> /noo/i.test('Noodle');`
	true

Error objects

Error objects are created either by the environment (the browser) or by your code.

```
> var e = new Error('jaavcsritp is _not_ how you spell it');
> typeof e;
"object"
```

Other than the `Error` constructor, six additional ones exist and they all inherit `Error`:

- `EvalError`
- `RangeError`
- `ReferenceError`
- `SyntaxError`
- `TypeError`
- `URIError`

The Error.prototype members

Property	Description

`name`	The name of the error constructor used to create the object:
	`> var e = new EvalError('Oops');`
	`> e.name;`
	"EvalError"

`message`	Additional error information:
	`> var e = new Error('Oops... again');`
	`> e.message;`
	"Oops... again"

JSON

The JSON object is new to ES5. It's not a constructor (similarly to `Math`) and has only two methods: `parse()` and `stringify()`. For ES3 browsers that don't support JSON natively, you can use the "shim" from `http://json.org`.

JSON stands for **JavaScript Object Notation**. It's a lightweight data interchange format. It's a subset of JavaScript that only supports primitives, object literals, and array literals.

Members of the JSON object

Method	Description
`parse(text, callback)`	Takes a JSON-encoded string and returns an object:

```
> var data = '{"hello": 1, "hi": [1, 2, 3]}';
> var o = JSON.parse(data);
> o.hello;
1
> o.hi;
[1, 2, 3]
```

The optional callback lets you provide your own function that can inspect and modify the result. The callback takes `key` and `value` arguments and can modify the `value` or delete it (by returning `undefined`).

```
> function callback(key, value) {
    console.log(key, value);
    if (key === 'hello') {
      return 'bonjour';
    }
    if (key === 'hi') {
      return undefined;
    }
    return value;
  }

>   var o = JSON.parse(data, callback);
hello 1
0 1
1 2
2 3
hi [1, 2, 3]
Object {hello: "bonjour"}
> o.hello;
"bonjour"
> 'hi' in o;
false
```

Method	Description

`stringify(value, callback, white)`	Takes any value (most commonly an object or an array) and encodes it to a JSON string.
	`> var o = {`
	`    hello: 1,`
	`    hi: 2,`
	`    when: new Date(2015, 0, 1)`
	`  };`
	
	`> JSON.stringify(o);`
	"{"hello":1,"hi":2,"when":"2015-01-01T08:00:00.000Z"}"
	The second parameter lets you provide a callback (or a whitelist array) to customize the return value. The whitelist contains the keys you're interested in:
	`JSON.stringify(o, ['hello', 'hi']);`
	"{"hello":1,"hi":2}"
	The last parameter helps you get a human-readable version. You specify the number of spaces as a string or a number.
	`> JSON.stringify(o, null, 4);`
	"{
	"hello": 1,
	"hi": 2,
	"when": "2015-01-01T08:00:00.000Z"
	}"

12
Regular Expressions

When you use regular expressions (discussed in *Chapter 4*, *Objects*), you can match literal strings, for example:

```
> "some text".match(/me/);
["me"]
```

But, the true power of regular expressions comes from matching patterns, not literal strings. The following table describes the different syntax you can use in your patterns, and provides some examples of their use:

Pattern	Description
[abc]	Matches a class of characters. `> "some text".match(/[otx]/g);` **["o", "t", "x", "t"]**

[a-z]	A class of characters defined as a range. For example, [a-d] is the same as [abcd], [a-z] matches all lowercase characters, [a-zA-Z0-9_] matches all characters, numbers, and the underscore character.
	`> "Some Text".match(/[a-z]/g);`
	["o", "m", "e", "e", "x", "t"]
	`> "Some Text".match(/[a-zA-Z]/g);`
	["S", "o", "m", "e", "T", "e", "x", "t"]

[^abc]	Matches everything that is not matched by the class of characters. `> "Some Text".match(/[^a-z]/g);` **["S", " ", "T"]**

Pattern	Description

a|b	Matches a or b. The pipe character means OR, and it can be used more than once.
	`> "Some Text".match(/t|T/g);`
	["T", "t"]
	`> "Some Text".match(/t|T|Some/g);`
	["Some", "T", "t"]

a(?=b)	Matches a only if followed by b.
	`> "Some Text".match(/Some(?=Tex)/g);`
	null
	`> "Some Text".match(/Some(?= Tex)/g);`
	["Some"]

a(?!b)	Matches a only when not followed by b.
	`> "Some Text".match(/Some(?! Tex)/g);`
	null
	`> "Some Text".match(/Some(?!Tex)/g);`
	["Some"]

\	Escape character used to help you match the special characters used in patterns as literals.
	`> "R2-D2".match(/[2-3]/g);`
	["2", "2"]
	`> "R2-D2".match(/[2\-3]/g);`
	["2", "-", "2"]

`\n`	New line
`\r`	Carriage return
`\f`	Form feed
`\t`	Tab
`\v`	Vertical tab
`\s`	White space, or any of the previous five escape sequences. `> "R2\n D2".match(/\s/g);` **["\n", " "]**
`\S`	Opposite of the above; matches everything but white space. Same as [^\s]: `> "R2\n D2".match(/\S/g);` **["R", "2", "D", "2"]**
`\w`	Any letter, number, or underscore. Same as [A-Za-z0-9_]. `> "S0m3 text!".match(/\w/g);` **["S", "0", "m", "3", "t", "e", "x", "t"]**

Pattern	**Description**
`\W`	Opposite of `\w`. `> "S0m3 text!".match(/\W/g);` **[" ", "!"]**
`\d`	Matches a number, same as [0-9]. `> "R2-D2 and C-3PO".match(/\d/g);` **["2", "2", "3"]**
`\D`	Opposite of `\d`; matches non-numbers, same as [^0-9] or [^\d]. `> "R2-D2 and C-3PO".match(/\D/g);` **["R", "-", "D", " ", "a", "n", "d", " ", "C", "-", "P", "O"]**

`\b`	Matches a word boundary such as space or punctuation.
	Matching R or D followed by 2:
	`> "R2D2 and C-3PO".match(/[RD]2/g);`
	["R2", "D2"]
	Same as above but only at the end of a word:
	`> "R2D2 and C-3PO".match(/[RD]2\b/g);`
	["D2"]
	Same pattern but the input has a dash, which is also an end of a word:
	`> "R2-D2 and C-3PO".match(/[RD]2\b/g);`
	["R2", "D2"]

`\B`	The opposite of `\b`.
	`> "R2-D2 and C-3PO".match(/[RD]2\B/g);`
	null
	`> "R2D2 and C-3PO".match(/[RD]2\B/g);`
	["R2"]

`[\b]`	Matches the backspace character.
`\0`	The null character.
`\u0000`	Matches a Unicode character, represented by a four-digit hexadecimal number. `> "стоян".match(/\u0441\u0442\u043E/);` **["сто"]**

`\x00`	Matches a character code represented by a two-digit hexadecimal number.
	`> "\x64";`
	`"d"`
	`> "dude".match(/\x64/g);`
	`["d", "d"]`

Pattern	**Description**
`^`	The beginning of the string to be matched. If you set the m modifier (multi-line), it matches the beginning of each line.
	`> "regular\nregular\nexpression".match(/r/g);`
	["r", "r", "r", "r", "r"]
	`> "regular\nregular\nexpression".match(/^r/g);`
	["r"]
	`> "regular\nregular\nexpression".match(/^r/mg);`
	["r", "r"]
`$`	Matches the end of the input or, when using the multiline modifier, the end of each line.
	`> "regular\nregular\nexpression".match(/r$/g);`
	null
	`> "regular\nregular\nexpression".match(/r$/mg);`
	["r", "r"]
`.`	Matches any single character except for the new line and the line feed.
	`> "regular".match(/r./g);`
	["re"]
	`> "regular".match(/r.../g);`
	["regu"]
`*`	Matches the preceding pattern if it occurs zero or more times. For example, `/.*/` will match anything including nothing (an empty input).
	`> "".match(/.*/);`
	[""]
	`> "anything".match(/.*/);`
	["anything"]
	`> "anything".match(/n.*h/);`
	["nyth"]
	Keep in mind that the pattern is "greedy", meaning it will match as much as possible:
	`> "anything within".match(/n.*h/g);`
	["nything with"]
`?`	Matches the preceding pattern if it occurs zero or one times.
	`> "anything".match(/ny?/g);`
	["ny", "n"]

Pattern	Description

+	Matches the preceding pattern if it occurs at least once (or more times).
	`> "anything".match(/ny+/g);`
	["ny"]
	`> "R2-D2 and C-3PO".match(/[a-z]/gi);`
	["R", "D", "a", "n", "d", "C", "P", "O"]
	`> "R2-D2 and C-3PO".match(/[a-z]+/gi);`
	["R", "D", "and", "C", "PO"]

{n}	Matches the preceding pattern if it occurs exactly n times.
	`> "regular expression".match(/s/g);`
	["s", "s"]
	`> "regular expression".match(/s{2}/g);`
	["ss"]
	`> "regular expression".match(/\b\w{3}/g);`
	["reg", "exp"]

{min,max}	Matches the preceding pattern if it occurs between a min and max number of times. You can omit max, which will mean no maximum, but only a minimum. You cannot omit min.
	An example where the input is "doodle" with the "o" repeated 10 times:
	`> "doooooooooodle".match(/o/g);`
	["o", "o", "o", "o", "o", "o", "o", "o", "o", "o"]
	`> "doooooooooodle".match(/o/g).length;`
	10
	`> "doooooooooodle".match(/o{2}/g);`
	["oo", "oo", "oo", "oo", "oo"]
	`> "doooooooooodle".match(/o{2,}/g);`
	["oooooooooo"]
	`> "doooooooooodle".match(/o{2,6}/g);`
	["oooooo", "oooo"]

(pattern)	When the pattern is in parentheses, it is remembered so that it can be used for replacements. These are also known as capturing patterns.
	The captured matches are available as $1, $2,... $9
	Matching all "r" occurrences and repeating them:
	`> "regular expression".replace(/(r)/g,  '$1$1');`
	"rregularr exprression"
	Matching "re" and turning it to "er":
	`> "regular expression".replace(/(r)(e)/g, '$2$1');`
	"ergular experssion"

Pattern	**Description**
(?:pattern)	Non-capturing pattern, not remembered and not available in $1, $2... Here's an example of how "re" is matched, but the "r" is not remembered and the second pattern becomes $1: `> "regular expression".replace(/(?:r)(e)/g, '$1$1');` **"eegular expeession"**

Make sure you pay attention when a special character can have two meanings, as is the case with `^`, `?`, and `\b`.

Biblography

This course is a blend of text and quizzes, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Mastering Javascript, Ved Antani*
- *Learning Object-Oriented Programming, Gastón C. Hillar*
- *Object-Oriented JavaScript - Second Edition , Stoyan Stefanov & Kumar Chetan Sharma*